LBNL-42877

ERNEST ORLANDO LAWRENCE
BERKELEY NATIONAL LABORATORY

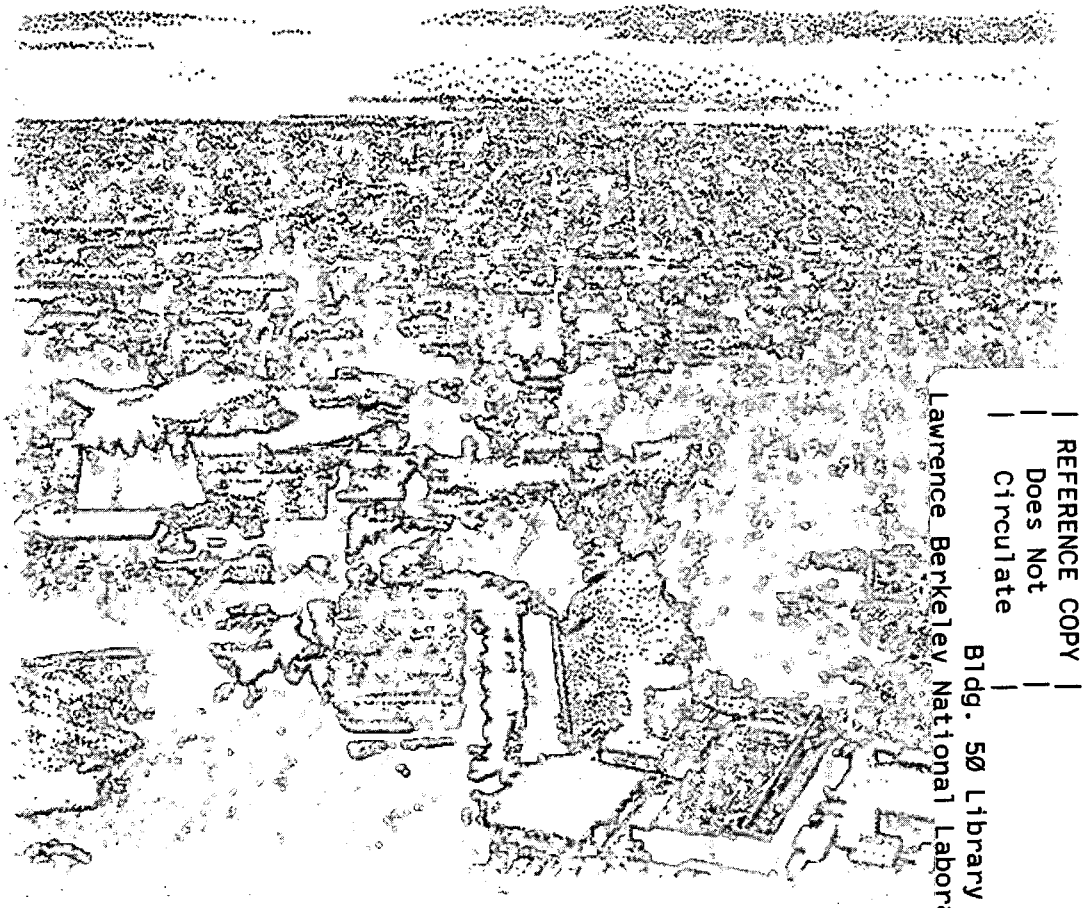
Femtosecond Photoelectron Spectroscopy: A New Tool for the Study of Anion Dynamics

Benjamin J. Greenblatt

Chemical Sciences Division

February 1999

Ph.D. Thesis



LBNL-42877

DISCLAIMER

LBNL-42877

Femtosecond Photoelectron Spectroscopy: A New Tool for the Study of Anion Dynamics

by

Benjamin Jefferys Greenblatt

B.S. (Haverford College) 1993

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Chemistry

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Daniel M. Neumark, Chair
Professor Ronald C. Cohen
Professor Jeffrey Bokor

Spring 1999

This work was supported in part by the Director, Office of Science, Office of Basic Energy Sciences, Chemical Sciences Division, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098, by the National Science Foundation under Grant No. CHE-9710243, and by the Defense University Research Instrumentation Program under Grant No. F49620-95-1-0078.

Femtosecond Photoelectron Spectroscopy: A New Tool for the Study of Anion Dynamics

by

Benjamin Jefferys Greenblatt

Abstract

Femtosecond Photoelectron Spectroscopy: A New Tool for the Study of Anion Dynamics

by

Benjamin Jefferys Greenblatt

Doctor of Philosophy in Chemistry

University of California, Berkeley

Professor Daniel M. Neumark, Chair

A new experimental technique for the time-resolved study of anion reactions is presented. Using femtosecond laser pulses, which provide extremely fast (~100 fs) time resolution, in conjunction with photoelectron spectroscopy, which reveals differences between anion and neutral potential energy surfaces, a complex anion reaction can be followed from its inception through the formation of asymptotic products. Experimental data can be modeled quantitatively using established theoretical approaches, allowing for the refinement of potential energy surfaces as well as dynamical models.

After a brief overview, a detailed account of the construction of the experimental apparatus is presented. Documentation of the data acquisition program is contained in the Appendix. The first experimental demonstration of the technique is then presented for I_2^- photodissociation, modeled using a simulation program which is also detailed in the Appendix. The investigation of I_2^- photodissociation in several size-selected $I_2^-(Ar)_n$ (n = 6-20) and $I_2^-(CO_2)_n$ (n = 4-16) clusters forms the heart of the dissertation. In a series of chapters, the numerous effects of solvation on this fundamental bond-breaking reaction

Abstract

are explored, the most notable of which is the recombination of I_2^- on the ground $\tilde{X}(^2\Sigma_u^+)$ state in sufficiently large clusters. Recombination and trapping of I_2^- on the excited $\tilde{A}(^2\Pi_{3/2,g})$ state is also observed in both types of clusters. The studies have revealed electronic state transitions, the first step in recombination, on a ~500 fs to ~10 ps timescale. Accompanying the changes in electronic state is solvent reorganization, which occurs on a similar timescale. Over longer periods (~1 ps to >200 ps), energy is transferred from vibrationally excited I_2^- to modes of the solvent, which in turn leads to solvent evaporation. These effects become more important as cluster size increases. In addition, differences in timescale and mechanism are observed between clusters of Ar, which binds to I^- and I_2^- rather weakly, and CO_2, whose large quadrupole moment allows substantially stronger binding to these anions.

Dedication

This dissertation is dedicated to my father

Raymond Benjamin Greenblatt

His unwavering love for me throughout my life

has allowed me to soar

Table of contents

Preface

As I have learned throughout my graduate career, scientific language is often extremely difficult for non-scientists to comprehend, and even when it is translated into common usage, the ideas themselves are sometimes a challenge. I have tried my hardest to provide here a short "layman's" explanation of my work, with the hope that, even if the rest of the dissertation is unintelligible, one can still walk away with a general understanding of the research, and where it fits into a larger context.

A femtosecond is an extremely short unit of time, a million times faster than a nanosecond, the time in which a computer can (currently) make decisions. It is so short, in fact, that light, which can circle the Earth seven times each second, only moves a hair's breadth. It is also shorter than the time it takes many molecules to vibrate. This means that the atoms of a molecule can in principle be observed in different positions at different times, rather than being smeared out over a range of locations if they were measured over many vibrations. We use these extremely short bursts of light as "strobe lights" for molecular motion. Everybody knows these devices from the dance floor: a quick burst of light which seems to freeze the motions of people and objects when observed by the eye.

The same approach is used to study molecules, employing a pair of light pulses to first, rapidly initiate a chemical reaction, and second, to take a "picture" of the reaction a short time later. We can use the dance floor as a metaphor for our experiment: the first flash signals a dancer to begin dancing, and the second flash is used to photograph her. The dancer will have to have good stamina, because she will be asked to repeat her moves precisely many thousands of times, while the time delay between pulses is slowly changed! Fortunately, molecules will always perform the same "dance" if they are care-

fully prepared beforehand, so molecular fatigue is not a concern here. In the end, a detailed movie of the dance is obtained. This approach is far more complex than simply filming the dance as it occurs, but the "movie camera" option is not possible for molecules: although the flashes of light happen very quickly, recording each "frame" takes far longer. Also, the second flash of light destroys the molecule!

We conduct our experiments inside a vacuum chamber, that is, a place where all the air has been removed. This is because collisions between the molecules under study and molecules of the surrounding air will invalidate our careful measurements, and cannot be tolerated. Typically, only one trillionth of atmospheric pressure is used, which sounds phenomenal, but it is actually routinely obtained for many scientific experiments done in vacuum. We produce a stream of negatively charged molecules ("anions") which are then guided from one end of the chamber to the other (a distance of 1.3 meters) using a series of small electric fields, rather like guiding a fast-moving pinball with paddles and bumpers. When the anions reach their destination, they are bathed with laser light for a few brief instants, after which "photoelectrons" are produced (photo = created from light; electron = a subatomic particle which orbits the nucleus of an atom). These electrons travel with different speeds toward a detector at the end of another tube (also about 1.3 meters long). The laser pulses start a clock which is stopped when the electrons are detected, revealing the time it took them to traverse the distance, and hence their speed. Speed is converted into a more useful quantity called "electron kinetic energy," and the data take the form of a histogram called a "spectrum," showing the numbers of electrons with each possible electron kinetic energy. While far from being a photograph of the molecule, the spectrum reveals important information about the molecule's structure, and

it is fairly easy to convert the information into a more intuitive physical picture. The study of a spectrum is called "spectroscopy."

The focus of my research is on understanding how a chemical reaction changes its course when it is surrounded by a liquid. Rather than studying the reaction in a liquid directly, we simulate a liquid environment by making small anion "clusters" inside the vacuum chamber, consisting of no more than a single layer of liquid molecules (about 16-20) surrounding the chemical reaction. I will not attempt to explain how the number of molecules in our clusters can be controlled, but suffice it to say that the number may be varied one at a time. We study the motion of the reaction without any liquid, then begin adding liquid molecules, and see how the motions change. In our case, the basic reaction is simply the breaking of a molecule into two atoms. In a cluster, however, collisions with the liquid molecules allow the atoms to come together again, forming the original molecule. The amount of "recombined" molecules increases as the size of the liquid cluster grows, until no atoms can escape the cluster.

The process by which the recombination process happens is very interesting: for the first few hundred to few thousand femtoseconds, the atoms appear to separate, but collisions between the atoms and the liquid heats the cluster, so that it begins to jiggle wildly. In order for the atoms to recombine, the jiggling must be reduced. This happens by expelling liquid molecules from the cluster, each of which takes away some of the heat. Over a period ranging from 10,000 femtoseconds to more than 200,000 femtoseconds, the cluster gradually cools, and the atoms recombine. The process of expelling liquid molecules is called evaporation, and is completely analogous to what happens to peo-

ple on a hot day: evaporating water from our skin cools us by taking away part of our heat.

Thus, there is the dissertation in a nutshell: using femtosecond photoelectron spectroscopy to study chemical reactions in small anion clusters.

Acknowledgments

Five and a half years is a long time to spend in one place, working on a single project, and I did not make it through alone.

I would like to acknowledge Dan Neumark for taking me into his group, for allowing me to be the first student on the FPES project, and for being my advisor.

Warm thanks go to Marty Zanni, my sole partner in the lab for three years. Marty's natural talent for experiments quickly became apparent, despite his year lag behind me, and we built the FPES machine as equals. Marty has a spontaneity to his scientific approach which was a welcome contrast to my more methodical style, and I believe we learned from each other how to be more effective experimenters. I also thank him for bringing a lot of fun into lab, with his music, swing dancing, chess games, midnight soccer matches (I only heard about them, not a player myself), cultural discussions, and general social coordination of the group.

Besides Marty, the FPES machine has seen a number of faces throughout my career. I acknowledge the visitations of Benoit Soep from the Université Paris-Sud, France; Rainer Weinkauf and Leo Lehr from the Technische Universität München, Germany; and Gustav Gerber from the Universität Würzburg, Germany. Current team members include graduate student Alison Davis, and our first post-doctoral student, Christian Frischkorn. Alison in particular I've gotten to know over the past year, as she spent a commendable amount of time in the lab from the beginning. My last weeks in lab, working on the first S_N2 reaction experiments, she almost always stayed as late as I did, even into the wee hours of the morning. I congratulate her on passing her second year exams, and wish her many successful experiments in the future.

Among other current group members, I extend a warm hand in thanks, both for practical assistance in the lab and as friends. In particular, I wish to acknowledge Mike Furlanetto for many discussions on a wide variety of scientific and cultural topics, and for being an excellent computer administrator during the last couple years. I also want to acknowledge Weizhong Sun for interesting discussions about China, and, especially, for arranging a lecture appearance for me at Beijing University with Professor Liming Ying, when I visited the country last summer.

Former Neumark group members are numerous, and I acknowledge them all for their direct and indirect gestures. In addition, I want to thank a few people I was especially close to. Among the graduate students, David Osborn was a role model for me from the beginning, as I was quickly awed by his multiple talents apart from science, and he made an effort to share some of them with me: rock climbing, river rafting, kite flying, carillon playing, and bread baking. Don Arnold gave me encouragement when I was first starting to think about the FPES machine (a year before it was constructed), and helped to orient me with his computer programs which I would later transform into the FPES data acquisition code. Cangshan Xu and I bonded in a number of ways: as computer afficianados, as tennis players, and as admirers of Chinese culture and language. Ivan Yourshaw also became close to me, as a movie hound, pop psychologist, environmentalist, feminist and fellow cynic.

There are also a few former post-docs I wish to highlight. Esther de Beer had stories of Europe to dazzle me with, and was my swimming partner for the last six months of her stay. Gordon Burton was a role model for me, always calm, positive, cultured, and fascinated by machines as well as science; he was a great brainstorming partner. David

Leahy, resident computer expert, spectroscopy guru, and comedian, tickled me with his off-the-wall humor, and taught me quite a bit about science. Finally, David Mordaunt became more of a friend after leaving the group, having discovered our similar temperaments and interests, and we have fallen into a social foursome together with our partners.

My contacts at Berkeley were not limited to the Neumark group, and there are several people outside the group I wish to acknowledge. For assistance with understanding femtosecond lasers and/or femtosecond photoelectron spectroscopy, Matt Blackwell and Pete Ludowise of the Chen group, Matt Asplund of the Charles Harris group, and Victor Batista of the Miller group all gave generously of their time. Matt and Pete, in particular, were practically coworkers for a time, both because their project strongly resembled ours (though they looked at neutral molecules), and because they were in our lab lending and borrowing equipment like any other group member. We hope our expertise was as beneficial to their project as theirs was to ours. I acknowledge student seminar program organizers Amy Herhold and C-J Lee of the Alivisatos group, Haw Yang of the Charles Harris group, and Linda Koch of the Cohen group: they all contributed to the success of the program, and with their sunny personalities, made it worthwhile for me to come to meetings.

There is another group of people whom I acknowledge for their innate abilities in speaking Chinese: Cangshan Xu and Weizhong Sun (Neumark group), C-J Lee (Alivisatos group), Haw Yang and Nien-hui Ge (Charles Harris group), Wenhong Yang and Hongwen Li (Strauss group), and Neil Fang (Accounting office). In addition, I acknowledge the patience of my Chinese teachers in the East Asian Languages Department: Joyce Wang, Ying Yang and Cecilia Chu. An interest of mine since college, I only had the op-

portunity to take courses in Chinese language during my fourth year at Berkeley. Once I had begun my studies, the above-named group of people received constant entreaties for spoken and written practice. The culmination of my studies in a sightseeing tour of China together with my parents in July 1998 was all the more enriching to me because of their help and encouragement.

Two faculty members deserve my special thanks for helping me choose a new career direction. Professor Ron Cohen has been patient, encouraging and generous of time and resources since I first expressed an interest in the field of atmospheric chemistry about a year ago. Since that time I have gone on to take a class with him, attend his group meetings, and ask him to sit on my dissertation committee. Dr. Susan Kegley has also served as an advisor of sorts, having spearheaded an inspiring, new environmental lab curriculum for Freshman Chemistry when I first came to Berkeley. Since that time, I have continued to keep tabs on her environmental projects, and she has encouraged me strongly in my recent pursuits of a post-doc in atmospheric chemistry.

I have had considerable professional interaction with two theoretical students from the University of Colorado, Boulder: James Faeder and Nicole Delaney of the Parson group. At first, we corresponded on the subject of anion cluster dynamics exclusively by phone and e-mail, but we finally met at the ACS Conference in Las Vegas in August 1997, and became friends. Their continued efforts in providing both calculation results and critical thinking has been invaluable to the writing process, and I wanted to acknowledge their good will.

I've had a few friends in the Bay Area completely outside the chemistry circle whom I wish to acknowledge. I met Lorin Gillin at Haverford College, and he has been in

and out of California ever since he graduated, but finally moved to San Francisco last year, where he has been pursuing high school science teaching. An avid outdoorsman, he and I have camped together in the mountains of Los Angeles, but haven't made it to the Sierra (yet). Ruth Wittman, an old family friend, first introduced me to Berkeley when I was here as a prospective student, and allowed me to stay in her home the following summer when I was shopping for an apartment. We get together occasionally for ice cream on the porch, and swap stories. My neighbors, Don and Linda Weingarten, share my love of science, and have employed me this past year and a half to tutor their two children, Eric and Neil. Don, especially, entertains me with his tall but true tales, and discussions on a wide variety of topics.

Out of state are some of my closest friends, and I acknowledge all of them for making the effort to keep in touch with me. John Kerrigan is a high school acquaintance, who became a more serious friend only after our graduations from college. He moved closer to me (to Las Vegas) in 1995, and subsequently we have visited each other several times along with his new wife, Kristin. An expert in Irish literature, John's intellect, humor and incomparable hospitality have helped sustain me through my years at Berkeley. Evan Manvel was another Haverford student who has always inspired me with his strong activist spirit. First he lived in Corvallis, Oregon pursuing local environmental issues, and I visited him several times there. Then he returned to the East Coast where he studied public policy at Harvard. Now he's back in Oregon doing activism, though we haven't seen much of each other lately. Lastly, I met Ameet Raval during a summer internship at the Princeton Geophysical Fluid Dynamics Laboratory in 1991. We quickly became good friends, and remain so today. He left atmospheric physics for psychology a few years af-

ter we met, and is now almost through his Ph.D. program at Temple University (Philadelphia, PA). Not only a great mental sparring partner, his spiritual and emotional sides have been a blessing to experience in a friend.

My parents, Ray and Sue, and my brother, Alex, all live near West Chester, PA. They have been a constant support for me, and vacations are seldom considered without including them: the feeling of renewal in seeing family is tremendous. My uncle, Bill Jefferys, is an astronomer at the University of Austin, Texas, and is in some way responsible for inspiring me to pursue the Ph.D., since he's the only other family member who has one. My grandmother, Ena Jefferys, lives in the tiny town of Waitsfield, VT, where I've managed to visit her three times since coming to Berkeley. I always enjoy the views from her porch, the smells of her kitchen, and her warm and witty manner. While at Berkeley, I've lost my three other grandparents, Ben and Claire Greenblatt (in 1995), and Bill Jefferys Sr. (in 1996). Fortunately, I saw them all often enough that I didn't feel I had lost touch with them when they died, though it was still very painful when I heard the news from my parents 3,000 miles away.

Finally, there is one last person who has been patiently waiting until the end of my acknowledgments for recognition, and that is Noreen Buyers. First secretary of the group, then girlfriend, now fiancée, she has transformed my life as well as helped me through the most difficult year of my Ph.D. program. My thanks to her are without end, as our journey unfolds together.

The research described in this dissertation was supported by the Director, Office of Science, Office of Basic Energy Sciences, Chemical Sciences Division, of the U.S.

Acknowledgments

Department of Energy under Contract No. DE-AC03-76SF00098. Additional funds were provided by the National Science Foundation under Grant No. CHE-9710243, and the Defense University Research Instrumentation Program under Grant No. F49620-95-1-0078.

Chapter 1. Introduction

For physical chemists, a primary motivation for experimental and theoretical work is to improve understanding of the process of making and breaking of chemical bonds. We focus on characterization of potential energy surfaces, which govern both structure and dynamics of molecular systems. Structure is best studied using so-called "frequency domain" laser techniques which, owing to the extremely narrow bandwidth of most tunable lasers, has enabled the complex energetic surfaces of a large number of systems to be characterized in phenomenal detail. If a sufficiently detailed and accurate surface can be produced, and the system has only a few degrees of freedom, then all the dynamics can be calculated from the potential energy surfaces. However, only a small region of the coordinate space of a potential energy surface is generally accessible from the minimum-energy configurations typically studied to derive structural information. In this case, molecules must be perturbed (through laser excitation, collisions in a molecular beam, or other means) in order to explore new regions of potential surfaces. In the frequency domain, such experiments are valuable if the end result of such a perturbation can be observed, such as a spectrum or kinetic energy distribution of product molecules, or if the transition state of a reaction can be detected through the broadening of transitions in the reactant spectrum, or a direct spectral signature of the transition state.

The wide availability of femtosecond pulsed lasers has added powerful new tools to the study of dynamics, as well as molecular structure, through observation of reactions in the "time domain." As the vibrational period of a typical diatomic molecule, such as I_2, is on the order of 50 fs, comparable to laser pulse durations, these lasers have the unique ability to prepare a molecular "wavepacket," or coherent superposition of vibrational lev-

els, and subsequently probe the molecule's evolution in time. In other words, the dynamics of a chemical reaction can be studied as it unfolds, not just at its starting or ending points. Numerous systems have been examined using a variety of pump-probe techniques, in both the condensed and gas phases, examining small and large molecules, neutral and charged species.[1-9]

One area where relatively little work has been done up until now is the time-resolved study of anion reactions. Anions play essential roles in the condensed phase, where they are basic to organic and biological reactions (e.g., nucleophilic substitution, oxidation-reduction, proton transfer). In the gas phase, they are important in the atmosphere (e.g. O_3^- and CO_3^- are intermediates in stratospheric anion chemistry,[10,11] and increased electron densities in the sunlit ionosphere is attributed to O^- photodetachment[12]). However, the gas phase is also an ideal environment for studying fundamental reactions, without the distortions to potential surfaces caused by the proximity of solvent molecules encountered in the condensed phase.

While many gas-phase anion systems have been explored in the frequency domain, only a handful have been studied in the time domain, most notably, the experiments on $I_2^-(Ar)_n$ and $I_2^-(CO_2)_n$ clusters by the Lineberger group,[13-19] and more recently, the femtosecond photoelectron spectroscopy of Au_3^- by Eberhardt and coworkers.[20] Thus, time-resolved studies of gas-phase anion reactions represent an enormous untapped area of research. Through studying anion reactions, we hope to gain a better understanding of the underlying chemical processes involved, including the timescales of product formation in photodissociation reactions (such as I_2^-[21,22] and I_3^-[23,24]), the formation of intermediate states on more complex potential surfaces (e.g., the gas-phase S_N2 reaction Cl^-

+ CH_3CN),[25] or the role of solvent in altering a reaction [e.g., the formation of dipole-bound anion states in $I^-(H_2O)_n$ clusters,[26,27] and the solvent-induced recombination of I_2^- in $I_2^-(Ar)_n$ or $I_2^-(CO_2)_n$ clusters (see Chapters 4-7)].

In the gas phase, anions are difficult to produce in large amounts compared with neutral molecules, making many traditional detection schemes (direct absorption, laser-induced fluorescence, multiphoton ionization) infeasible. Sensitive methods have been developed for studying anions in the gas phase, most notably photoelectron spectroscopy. It is basically this technique, with a long history in the Neumark group, which has been coupled to a femtosecond pulsed laser, to enable the study of time-resolved anion reactions.

Photoelectron spectroscopy refers to a collection of related approaches for examining the electronic structure of gas-phase or surface-bound ions and molecules. Its origins lie in the discovery of the photoelectric effect, which identified a threshold photon energy for ejection of electrons from a metal surface.[28,29] As a gas-phase technique, its development in the early 1960s[30,31] has led to an explosive growth of the field, and photoelectron spectroscopy is now routinely used to study a wide variety of molecular and ionic systems.

The photoelectric effect can be summarized in terms of an energy conservation equation as follows:[32,33]

$$h\nu = EA + E_{\text{internal}} + E_{e^-} \qquad (1)$$

where $h\nu$ is the photon energy, EA is the electron affinity (or, in the case of a neutral molecule, it is replaced by IP, the ionization potential), E_{internal} is the internal energy of the newly-created neutral molecule (or positive ion), and E_{e^-} is the kinetic energy of the

electron. $E_{internal}$ can be further broken down into various components, e.g., electronic, vibrational and rotational. Since the electron produced is free, all its energy is kinetic; thus useful implementation of the principle involves measuring either the kinetic energy of ejected electrons using a fixed-frequency photon, or varying the photon energy while monitoring electrons ejected with a specific (usually zero) kinetic energy. Both methods have been implemented successfully in the Neumark group laboratories, and each approach has different strengths and weaknesses. With a fixed-frequency laser source, an entire spectrum may be collected at once (photoelectron spectroscopy or PES),[34] but resolution is limited to ~50-100 cm^{-1} by the collection angle of the electron detector, the length of the flight tube, and the spatial extent of the ion and laser beams. When the laser frequency is tunable, the resolution may approach that of the laser, e.g. ~0.05 cm^{-1}, if electrons with nearly zero kinetic energy are detected [threshold or zero electron kinetic energy (ZEKE) spectroscopy],[35] but such electrons must have the proper angular momentum to be detected efficiently near threshold, and data collection is both slower and at a single energy at a time.

The extension of photoelectron spectroscopy to the ultrafast regime began with a series of experiments in the early 1990s studying intramolecular vibrational energy redistribution in gas-phase anilines,[36,37] using a pair of picosecond pulses to first promote molecules to an electronically excited state, and then to ionize them, producing photoelectrons. Both PES and ZEKE techniques were employed. Later studies utilizing femtosecond pulses examined the time-resolved vibrations of excited Na_3,[9] and I_2;[38] rates of internal conversion in hexatriene,[3] and the control of NO product states through the

shifting of potentials using intense femtosecond pulses,[39] using either ZEKE spectroscopy (in the first two examples), or PES.

Femtosecond photoelectron spectroscopy (FPES) for anions is similar to the above approaches for neutrals. It involves the following pump-probe scheme, using the photodissociation of a generic diatomic anion "XY^-" as an example:

$$XY^- \xrightarrow{h\nu_1} [XY^-]^* \xrightarrow{\Delta t} X\ldots Y^- \xrightarrow{h\nu_2} \begin{cases} XY + e^- \\ X + Y + e^- \end{cases} \tag{2}$$

where $[XY^-]^*$ is the anion in an excited electronic state immediately after excitation, $X\ldots Y^-$ is the anion at a later time, where the X and Y^- products have mostly separated, e^- is the detached photoelectron, $h\nu_1$ and $h\nu_2$ are femtosecond-duration pump and probe laser pulses, respectively, and Δt is a variable time delay between the pulses. The measured signal is the photoelectron, which is energy-analyzed by time-of-flight; therefore, an entire photoelectron spectrum may be collected for each pump-probe time delay Δt. .

The technique is illustrated in more detail in Fig. 1. Preparation of XY^- is assumed to be vibrationally cold, however, the same analysis may apply for a vibrationally hot ion, though dynamics will be blurred out due to a less localized starting wavepacket. Promotion to an anion excited state takes place with the pump photon. The wavepacket then evolves toward larger internuclear distances, i.e. dissociation into $X + Y^-$. Because of the ultrafast time duration of the laser pulses, this wavepacket motion can be followed with the probe pulse at various time delays between pump and probe pulses, indicated by Δt_1 and Δt_2. This second pulse detaches an electron from the anion, leaving the remaining neutral molecule in a state determined by the Franck-Condon overlap between anion and neutral wavefunctions. From Eq. 1, the kinetic energy of the electron is determined by the

internal energy of the neutral molecule; this is reflected in the figure by horizontal lines representing vibrational eigenstates of the neutral, which connect to peaks in the photoelectron spectrum on the right. The total energy of the probe photon corresponds to the origin of the spectrum.

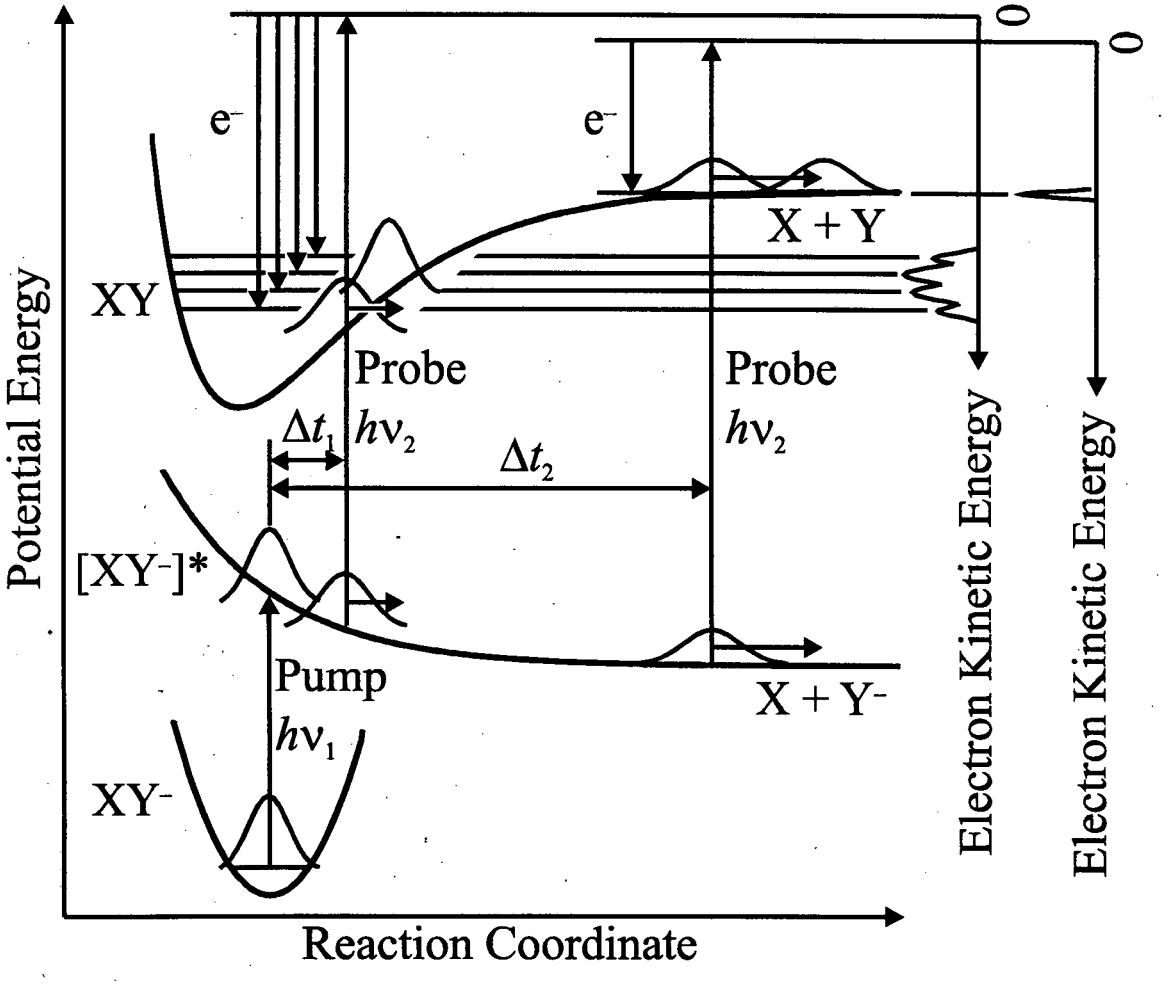


Fig. 1. Schematic diagram of the FPES technique.

At time delay Δt_1, the wavepacket is still on the repulsive wall of the anion potential, so overlap with the neutral potential is strongest in the bound region, generating an extended vibrational progression in the photoelectron spectrum. By time delay Δt_2, the wavepacket has reached the dissociation asymptote, and both potentials are flat; the pho-

toelectron spectrum displays a single sharp peak, corresponding to the energy of Y^- photodetachment. The horizontal arrow near each wavepacket represents its average kinetic energy along the reaction coordinate, and has the effect of shifting the Franck-Condon overlap toward the turning point (zero kinetic energy) region of the neutral potential.

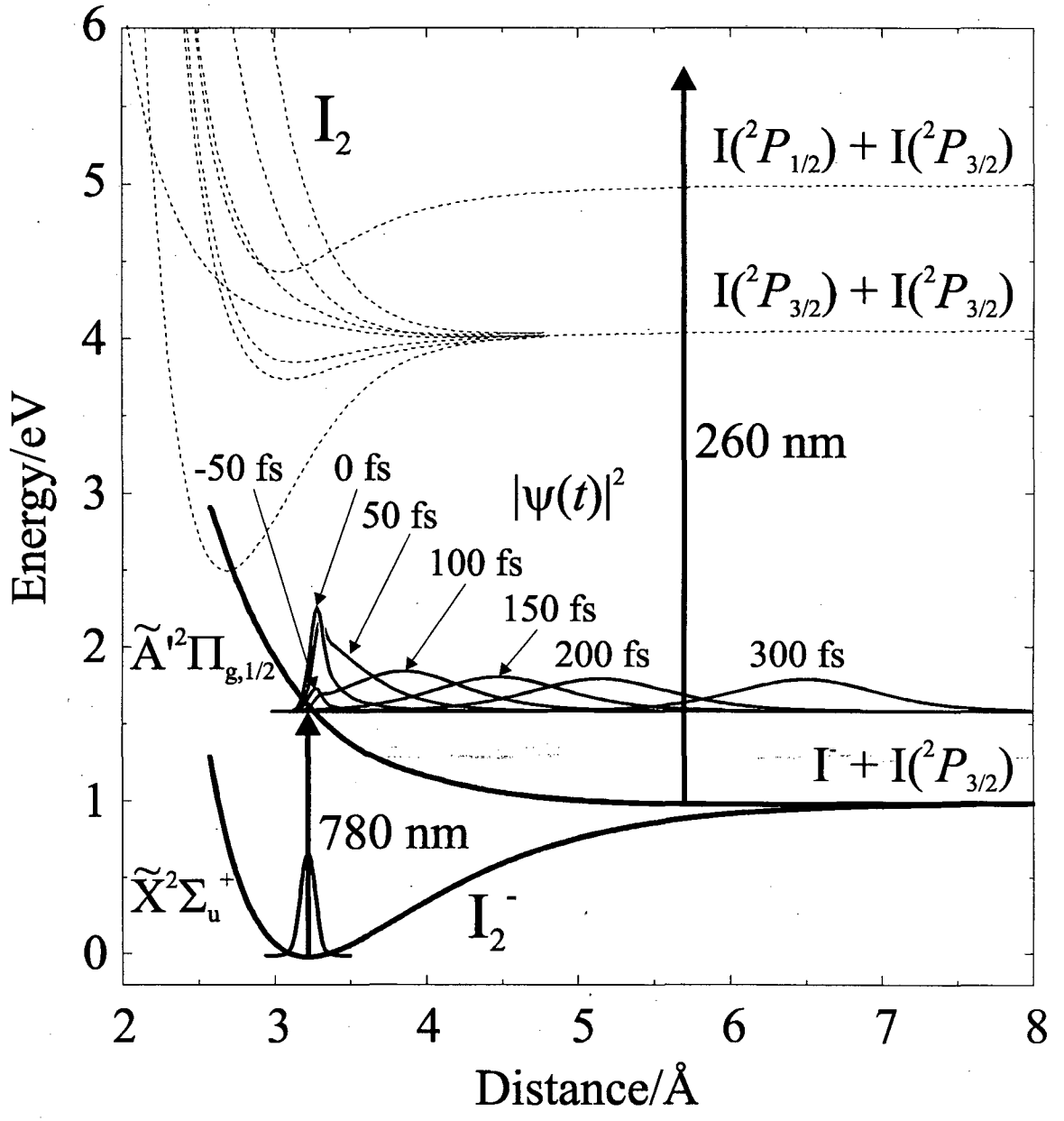
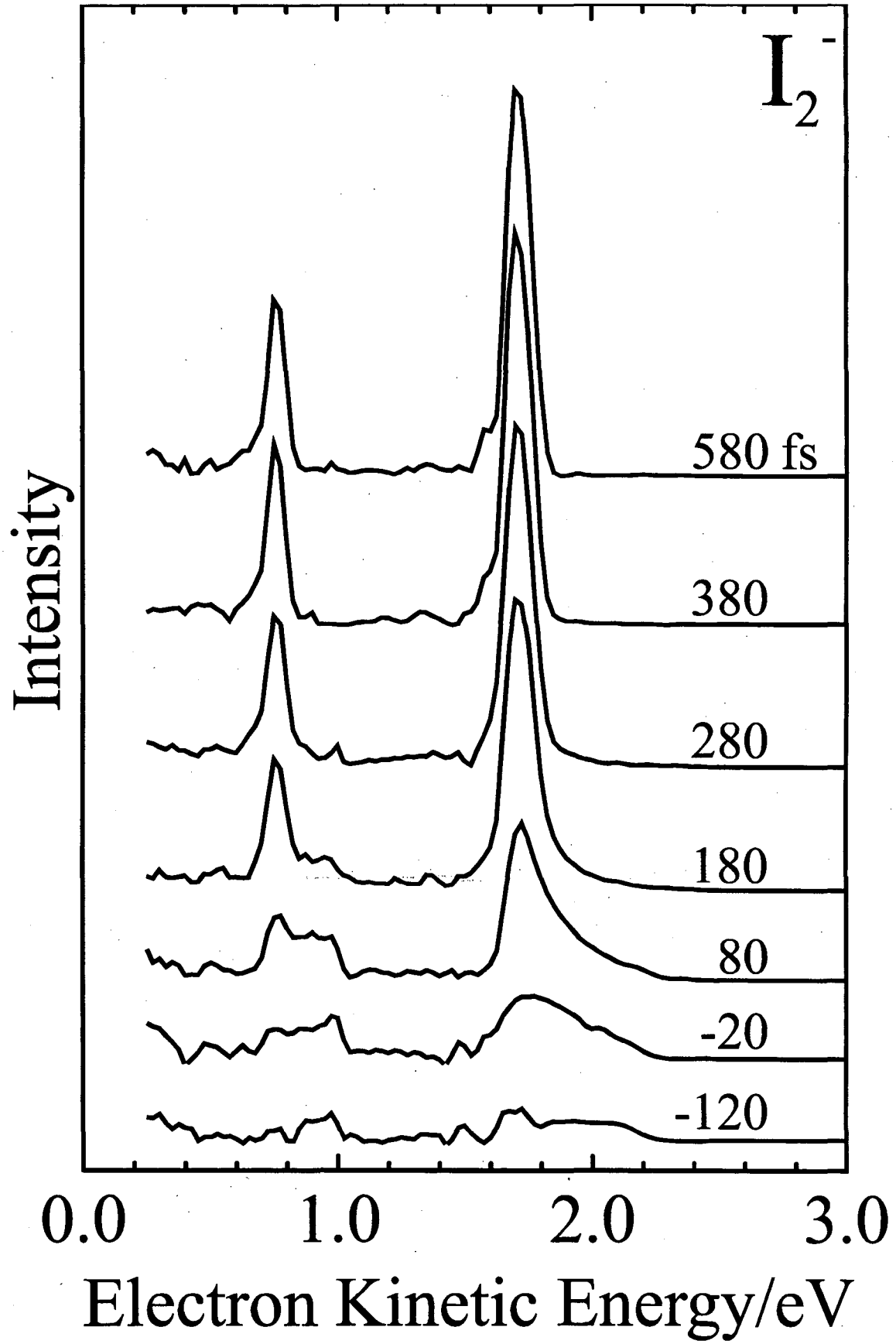


Fig. 2. Snapshots of calculated I_2^- wavepacket on the $\tilde{A}'(^2\Pi_{1/2,g})$ state.

As an example of a real system, a series of "snapshots" of the calculated evolving wavepacket for I_2^- photodissociation are shown in Fig. 2. Here the anion is excited from the ground $\tilde{X}(^2\Sigma_u^+)$ state to the excited $\tilde{A}'(^2\Pi_{1/2,g})$ state with 780 nm light of 80 fs duration. As the wavepacket moves considerably over the pump pulse duration, the wavepacket at 50 fs is highly asymmetric, with a strong peak in the initial Franck-Condon region, and a broad tail at larger internuclear distances. By 150 fs, the wavepacket is completely outside the Franck-Condon region, and by ~300 fs, it has reached the $I + I^-$ dissociation asymptote.

A set of experimental photoelectron spectra of this reaction, taken from Chapter 6, is shown in Fig. 3. At early time delays, the spectrum is dominated by intensity in two broad regions, roughly 0.7-1.0 eV, and 1.6-2.3 eV, corresponding to transitions to the two manifolds of neutral states correlating with the $^2P_{1/2}$ and $^2P_{3/2}$ asymptotes of I, respectively (see Fig. 2). By 380 fs, these broad features have been replaced by sharp features at 770 meV and 1.71 eV, respectively: transitions to the I spin-orbit states. As demonstrated in Zanni *et al.*,[22] these photoelectron spectra can be modeled quantitatively, confirming that wavepacket dynamics may be followed from start to finish over a duration of only a few hundred fs, and there is a considerable amount of information to be had from such spectra. The femtosecond photoelectron spectrum simulation program used to generate such spectra, including a more extensive theoretical background, is covered in Appendix 2.

Fig. 3. Experimental photoelectron spectra of I_2^- photodissociation (on next page).


I_2^-
580 fs
380
280
180
80
-20
-120
Intensity
0.0
1.0
2.0
3.0
Electron Kinetic Energy/eV

This dissertation contains, in addition to the first account of the FPES technique applied to anions (Chapter 3), a series of papers exploring the effects of solvent molecules on a chemical reaction (Chapters 4-7). Using mass-selected cluster anions to "tune" the amount of solvation one molecule at a time, the photodissociation of I_2^- in both Ar and CO_2 clusters has been studied. These experiments are based on earlier work by the Lineberger group, which examined anionic photofragment distributions of these clusters, as well as time-resolved two-photon absorption.[13-19] The goal of the FPES experiments has been to understand the evolution of the dynamics from the uncaged to fully-caged size regime. The level of detail afforded by measuring complete photoelectron spectra, rather than probing a specific transition of the anion, is unsurpassed in these reactions, and has allowed us to glimpse a much more complete picture than what was previously possible.

Although the detailed findings can be found in the individual chapters, a few key results are summarized here. With only a few (~4-6) solvent molecules present, photodissociation proceeds relatively unimpeded, though the effect of solvent on the motion of the iodine atoms is unmistakable. In larger clusters, interesting new dynamics occur, most importantly, the "caging" of the I^- and I atoms by the solvent to reform I_2^-, either on the ground state, or in an electronically excited state [$\tilde{A}(^2\Pi_{3/2,g})$]. By the time a full solvent shell is reached (20 for the case of Ar, or 16 for CO_2), 100% caging is observed, with a considerable speedup in the rate of I_2^- formation as well. Extensive vibrational relaxation of the I_2^- in the ground state is observed, accompanied by evaporation of solvent molecules to dissipate the energy. The main differences in the dynamics between the $I_2^-(Ar)_n$ and $I_2^-(CO_2)_n$ clusters is the presence of solvent-induced electronic transitions in the first

~500 fs-1 ps in $I_2^-(CO_2)_n$ clusters which are absent in $I_2^-(Ar)_n$ clusters, and considerably faster I_2^- formation and vibrational relaxation in $I_2^-(CO_2)_n$ clusters. These distinctions highlight the significantly stronger binding energy of CO_2 with I^- and I_2^- over that of Ar.

References

1 D. E. Smith and C. B. Harris, J. Chem. Phys. **87,** 2709 (1987).

2 C. Lienau and A. H. Zewail, J. Phys. Chem. **100,** 18629 (1996).

3 D. R. Cyr and C. C. Hayden, J. Chem. Phys. **104,** 771 (1996).

4 R. M. Bowman, M. Dantus, and A. H. Zewail, Chem. Phys. Lett. **161,** 297 (1989).

5 M. Gruebele, G. Roberts, M. Dantus, R. M. Bowman, and A. H. Zewail, Chem. Phys. Lett. **166,** 459 (1990).

6 M. Gruebele and A. H. Zewail, J. Chem. Phys. **98,** 883 (1993).

7 T. Baumert, V. Engel, C. Meier, and G. Gerber, Chem. Phys. Lett. **200,** 488 (1992).

8 T. Baumert, V. Engel, C. Rottgermann, W. T. Strunz, and G. Gerber, Chem. Phys. Lett. **191,** 639 (1992).

9 T. Baumert, R. Thalweiser, and G. Gerber, Chem. Phys. Lett. **209,** 29 (1993).

10 E. E. Ferguson, F. C. Fehsenfeld, and D. L. Albritton, in *Gas Phase Ion Chemistry*, Vol. 1, edited by M. T. Bowers (Academic Press, New York, 1979), pp. 45-82.

11 F. Arnold, in *Atmospheric Chemistry*, edited by E. D. Goldberg (Springer-Verlag, New York, 1982), pp. 273-300.

12 J. R. Peterson, J. Geophys. Res. **81,** 1433 (1976).

13 D. Ray, N. E. Levinger, J. M. Papanikolas, and W. C. Lineberger, J. Chem. Phys. **91,** 6533 (1989).

14 J. M. Papanikolas, J. R. Gord, N. E. Levinger, D. Ray, V. Vorsa, and W. C. Lineberger, J. Phys. Chem. **95,** 8028 (1991).

15 J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, J. R. Gord, and W. C. Lineberger, J. Chem. Phys. **97,** 7002 (1992).

16 J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, H. K. Buchenau, and W. C. Lineberger, J. Chem. Phys. **99,** 8733 (1993).

17 V. Vorsa, Ph.D. Thesis, University of Colorado, Boulder (1996).

18 V. Vorsa, P. J. Campagnola, S. Nandi, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **105,** 2298 (1996).

19 V. Vorsa, S. Nandi, P. J. Campagnola, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **106,** 1402 (1997).

20 G. Gantefor, S. Kraus, and W. Eberhardt, J. Electron Spectroscopy and Related Phenomena **88-91,** 35 (1998).

21 B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Chem. Phys. Lett. **258,** 523-529 (1996).

22 M. T. Zanni, V. S. Batista, B. J. Greenblatt, W. H. Miller, and D. M. Neumark, J. Chem. Phys. **110,** 3748 (1999).

23 M. T. Zanni, B. J. Greenblatt, A. V. Davis, and D. M. Neumark, in preparation .

24 M. T. Zanni, B. J. Greenblatt, A. V. Davis, and D. M. Neumark, Proc. SPIE **3271,** 196 (1998).

25 A. V. Davis, M. T. Zanni, and D. M. Neumark, in progress .

26 L. Lehr, M. T. Zanni, C. Frischkorn, R. Weinkauf, and D. M. Neumark, Science, in

press .

[27] M. T. Zanni, L. Lehr, B. J. Greenblatt, R. Weinkauf, and D. M. Neumark, Proceedings of the XIth Ultrafast Conference, Garmisch-Partenkirchen, Germany, in press .

[28] A. Einstein, Ann. Phys. **17,** 132 (1905).

[29] R. A. Millikan, Phys. Rev. **7,** 355 (1916).

[30] M. I. Al-Joboury and D. W. Turner, J. Chem. Phys. **37,** 3007 (1962).

[31] F. I. Vilesov, B. L. Kurbatov, and A. N. Terenin, Sov. Phys. Dokl. **6,** 490 (1961).

[32] J. H. D. Eland, *Photoelectron spectroscopy: an introduction to ultraviolet photoelectron spectroscopy in the gas phase*, 2nd ed. (Butterworths, Boston, 1984).

[33] J. W. Rabalais, *Principles of Ultraviolet Photoelectron Spectroscopy* (Wiley, New York, 1977).

[34] A. Weaver, Ph.D. Thesis, University of California, Berkeley (1991).

[35] T. N. Kitsopoulos, Ph.D. Thesis, University of California, Berkeley (1992).

[36] X. Song, C. W. Wilkerson, Jr., J. Lucia, S. Pauls, and J. P. Reilly, Chem. Phys. Lett. **174,** 377 (1990).

[37] J. M. Smith, X. Zhang, and J. L. Knee, J. Chem. Phys. **99,** 2550 (1993).

[38] I. Fischer, D. M. Villeneuve, M. J. J. Vrakking, and A. Stolow, J. Chem. Phys. **102,** 5566 (1995).

[39] P. Ludowise, M. Blackwell, and Y. Chen, Chem. Phys. Lett. **258,** 530 (1996).

Chapter 2. Experimental Apparatus

The femtosecond photoelectron spectrometer (FPES machine) is designed to collect time-dependent photoelectron spectra of anions. To accomplish this task, two fairly independent segments are coupled together: 1) a high-vacuum chamber, in which anions are produced and photoelectron signals detected; and 2) a femtosecond laser system, for producing and characterizing pulses of light at multiple wavelengths. Each is a highly complex apparatus, and will be described separately. Acquisition of data will be discussed last.

1. Vacuum system

The vacuum system consists of several differentially pumped regions, a drawing of which is shown in Fig. 1. Each region of the vacuum is labeled according to function and/or order of differential pumping: source, "zeroeth" differential, first differential, second differential and detector. Within the source region, anions are produced and cooled by crossing an electron beam with a supersonic gas expansion. Passing through a beam skimmer into the zeroeth differential region, the anions are extracted into a Wiley-McLaren[1] time-of-flight mass spectrometer. The first and second differential regions contain ion steering and focusing optics for controlling the position of the beam. Once inside the detector region, anions are intercepted by the laser pulses, which enter and exit the region through a pair of windows. After the anions have interacted with the laser, photoelectrons, neutrals, and possibly photofragment ions are produced. Photoelectrons are collected in a "magnetic bottle" time of flight energy analyzer, and detected with an electron detector at the end of a long flight tube. Ions or neutrals are detected by a

retractable detector, and photofragment ions are analyzed in an off-axis reflectron/detector. Prior to laser interaction, anions may also be mass-selected and/or decelerated with a mass gate/pulsed ion decelerator, in order to improve photoelectron energy resolution.

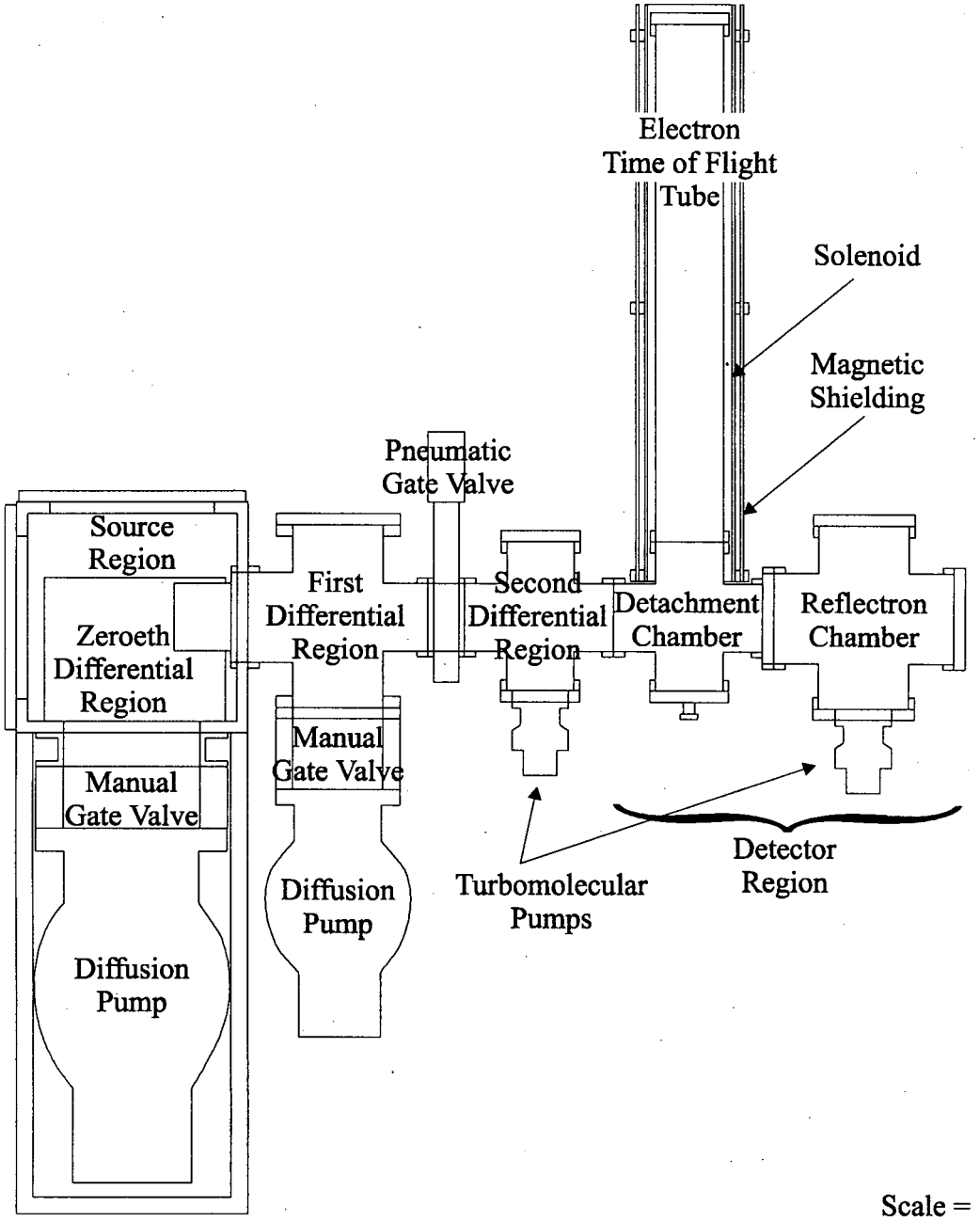


Fig. 1. FPES vacuum apparatus.

Chapter 2

1.1. Pumps

The design considerations of the FPES machine are rather unusual. Pulsed ion beams have been shown to work well for photoelectron spectroscopy, but in order to take advantage of the high repetition rate (1 kHz) of the femtosecond laser, anions must be produced at a comparable rate. This introduces complications to the pumping requirements of the system, as previous pulsed experiments in the Neumark Group run at 100 Hz or below. Each anion pulse generated involves the admission of a quantity of gas into the vacuum chamber, increasing the pressure, so the restriction to a working pressure of $\leq 5\times10^{-4}$ torr (both to prevent the diffusion pumps from stalling, and to maintain a viable electron beam) places serious constraints on the choice of primary pumps. Although the amount of gas contained in each pulse can be limited by reducing the hole diameter of the gas inlet valve, a minimum quantity is still required to produce reasonable anion signals. The solution has been found in a combination of large pumps, small valve orifices, and an "extra" differential region (typical vacuum systems built by the group have only four differential regions) to reduce the effects on the rest of the system of an unusually large gas pressure in the source region.

The source region is pumped by a Varian VHS-10 diffusion pump (4400 L/s pumping speed), backed by an Alcatel 2063 direct drive mechanical pump [50 cfm (cubic feet per minute) or 23.6 L/s]. When present, the zeroeth differential region is pumped by a second Varian VHS-10, backed by a Sargent-Welch belt-drive mechanical pump (~10 L/s). Otherwise, both Varian pumps evacuate the source region (backed by the Alcatel pump), for a total pumping speed of 8800 L/s. Maximum working pressure in the source

region is limited to approximately 5×10^{-4} torr, above which the diffusion pumps begin to stall, an undesirable (and messy) situation.

The first differential region is pumped by a Varian VHS-6 diffusion pump (1900 L/s), backed by an Alcatel 2021 direct drive mechanical pump (15 L/s). Typical working pressure in this region is 1×10^{-6} torr. The second differential and detector regions are each pumped by a Varian V-250 turbomolecular pump (250 L/s) and backed by a Varian SD-300 direct drive mechanical pump (5 L/s). The second differential region maintains a working pressure of 2×10^{-8} torr, while the detector region maintains 1×10^{-9} torr.

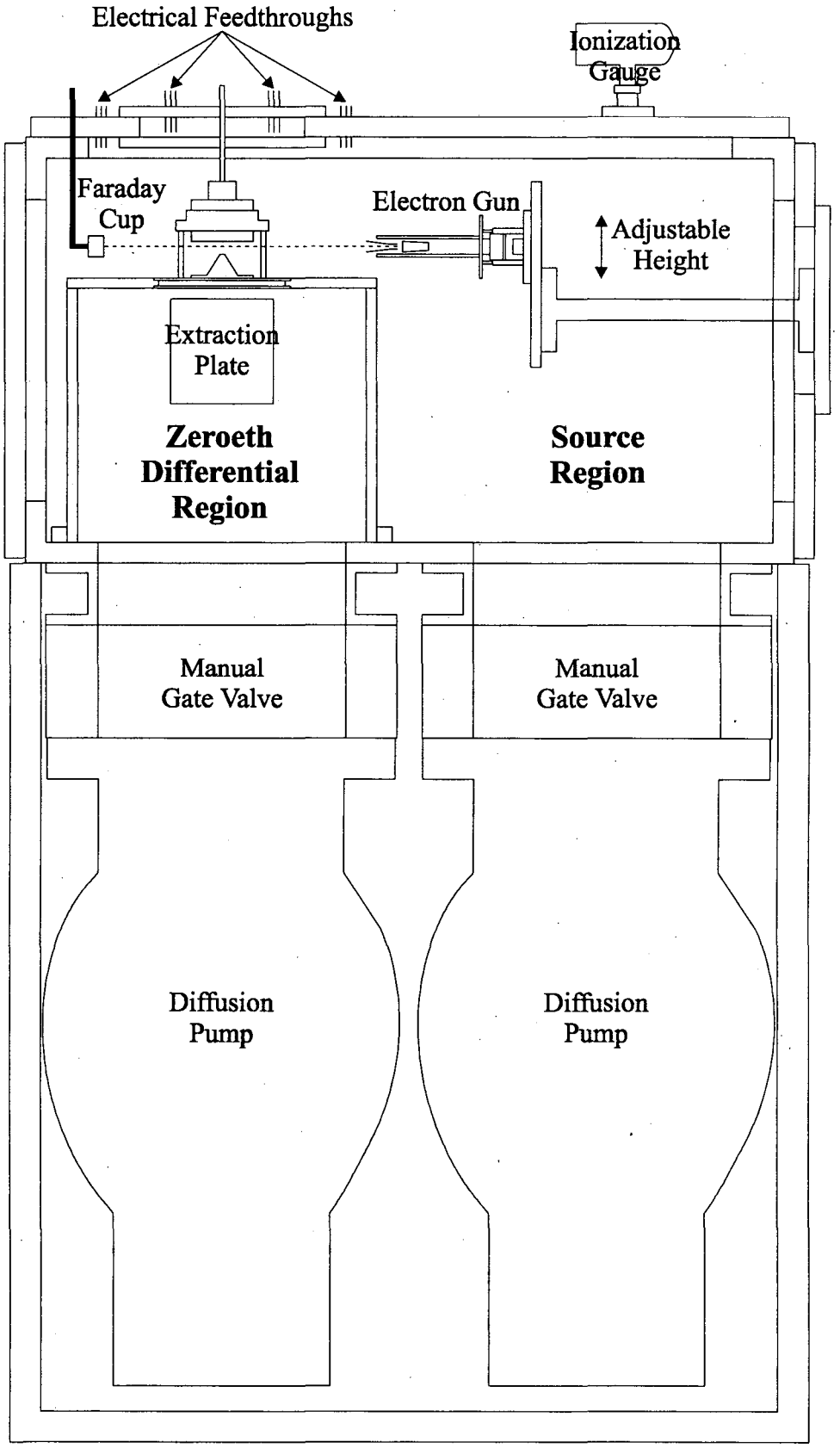
Between each region is a 3 mm diameter hole (between the source and zeroeth differential regions, the hole is effected by a skimmer, whose diameter can be varied), which enables differential pumping. An electropneumatic gate valve separates the first and second differential regions. The source, zeroeth differential and first differential regions are referred to as the low vacuum region, and generally do not attain pressures below 1×10^{-8} torr because of the use of rubber o-ring seals and a generally oily environment. The second differential and detector regions constitute the ultrahigh vacuum region, which routinely attains 5×10^{-10} torr. All seals in this region use copper gaskets, and outgassing materials are kept to a minimum: a capacitor for each detector, and few resistors associated with the deceleration and reflectron stacks. Any handling of materials to reside inside these regions must be done under extremely clean conditions, as a small amount of grease or dirt can spoil the ultrahigh vacuum environment. This region is usually baked for 1-3 days after initial pump-down.

1.2. Interlock

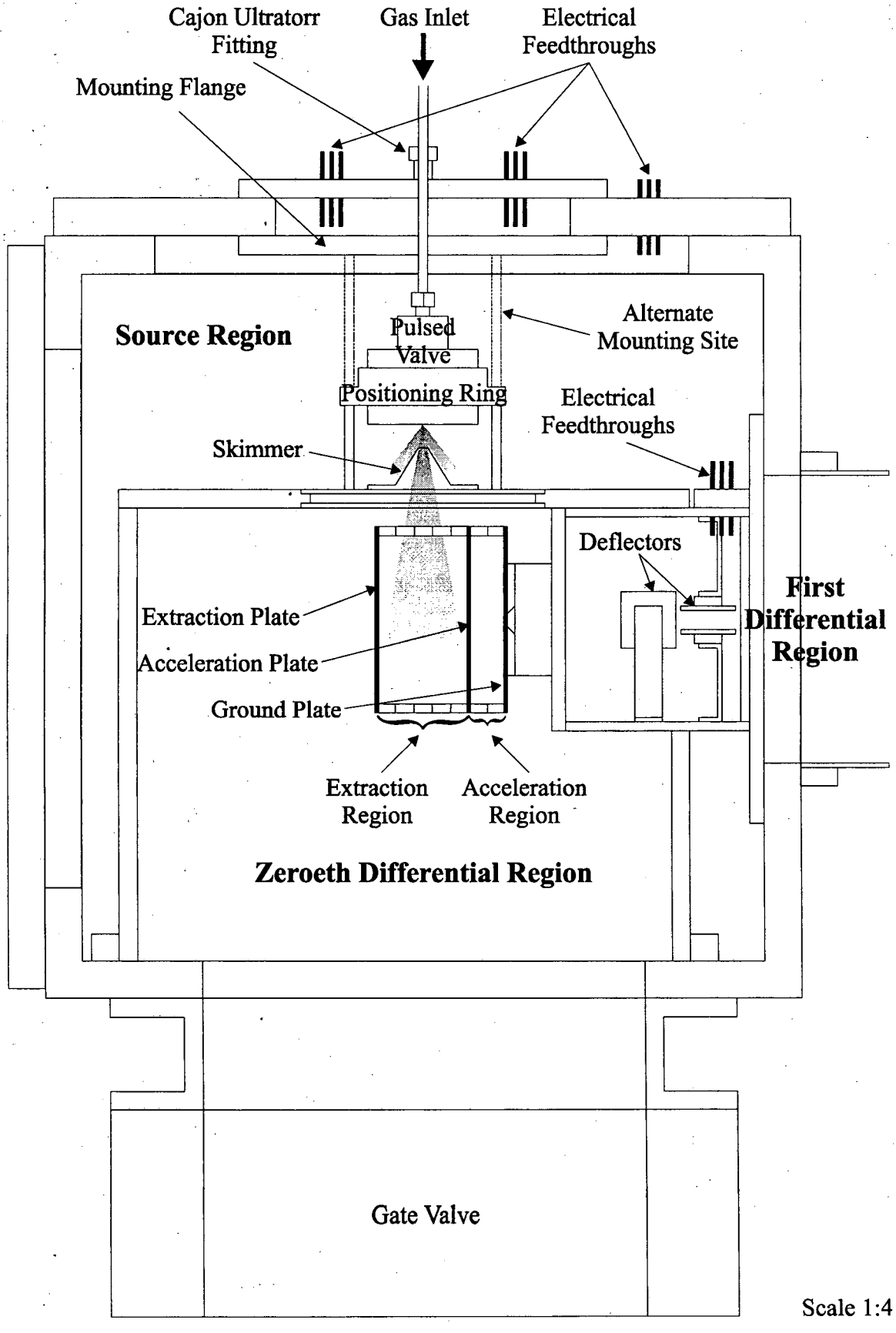
An automatic protection or "interlock" circuit is required for the automatic shutdown of the machine under unsafe operating conditions. Although a detailed description of the circuit will appear in the dissertation of Martin Zanni, who designed and built it, a brief summary of its functions is presented here. The interlock controls the power to the diffusion and turbomolecular pumps, electropneumatic valves and high-voltage equipment. Sensors including pressure gauges, temperature thermocouples (on the diffusion pumps) and speed regulators (on the turbomolecular pumps) enable the interlock circuit to safely turn off and isolate parts of the vacuum chamber, in order to protect sensitive equipment from damage, and neutralize any potentially dangerous (high temperature and/or pressure) situation. In addition, the vacuum system was designed so that, in the event of a power failure, key electropneumatic valves automatically vent the ultrahigh vacuum region to prevent the diffusion of mechanical pump oil into ultraclean regions. Without human intervention, when power is restored, the interlock keeps the system shut down since sensors then report an inoperable condition.

Fig. 2. Source and zeroeth differential regions – front view (on next page).

Fig. 3. Source and zeroeth differential regions – side view (on page 20).


Electrical Feedthroughs
Ionization Gauge
Faraday Cup
Electron Gun
Adjustable Height
Extraction Plate
Zeroeth Differential Region
Source Region
Manual Gate Valve
Manual Gate Valve
Diffusion Pump
Diffusion Pump

Scale 1:8


Cajon Ultratorr Fitting
Gas Inlet
Electrical Feedthroughs
Mounting Flange
Source Region
Pulsed Valve
Alternate Mounting Site
Positioning Ring
Electrical Feedthroughs
Skimmer
Deflectors
First Differential Region
Extraction Plate
Acceleration Plate
Ground Plate
Extraction Region
Acceleration Region
Zeroeth Differential Region
Gate Valve

Scale 1:4

1.3. Source and zeroeth differential regions

The source and zeroeth differential regions are shown in two views, from the front in Fig. 2 and from the side in Fig. 3. The source region consists of a large, stainless steel box with two solid sides (bottom and rear) and four open sides, on the outside of which are mounted removable aluminum doors. This design both saves on the total weight of the chamber, and offers maximum flexibility for modifications and access. Two 12" diameter manual gate valves (Kurt Lesker) are bolted to the bottom of the source chamber, and under each of these is bolted a diffusion pump. The left side of the chamber (as viewed from the front, looking toward the detector region) has holes drilled on the inside bottom and rear surfaces for mounting of a removable zeroeth differential region, which is described separately below. The right door of the chamber holds the electron gun flange, containing electrical feedthroughs and, on the inside, pairs of tapped holes for mounting and adjusting the height of the electron gun. The top door has several flanges for mounting the pulsed valve, Faraday cup, two sets of electrical feedthroughs, and an ionization gauge. Some electronic instrumentation (primarily the extraction/acceleration circuit) sits permanently on the top side of this door. The front door has a transparent acrylic flange for viewing inside the chamber while under vacuum. The left door is currently blank.

1.3.1. Pulsed valve

The pulsed valve, used to introduce gas into the vacuum chamber, is very similar in design to that used by the group's Fast Radical Beam Machine (FRBM),[2] and is originally based on the design of Trickl and Proch.[3] Briefly, it consists of a stainless steel cylindrical housing with coupling to the gas supply tube on the rear side, and an area on

the front for mounting a face plate with a variable diameter orifice. Inside the valve is a piezoelectric disk translator with a hole in the center, through which is mounted a poppet shaft. This seals against the inside surface of the faceplate with an o-ring when the valve is off, and is pulled back by the piezoelectric disk when voltage is applied, allowing gas to escape through the orifice into the source region.

The valve body itself is made up of two sections bolted together and sealed by an o-ring, and the faceplate, piezoelectric disk and poppet are all mounted to the front section. This scheme allows the alignment of the poppet with the faceplate to be adjusted without the valve in place in the vacuum chamber; the front section can be mounted on a flange elsewhere on a source chamber door and adjusted from outside, while the front of the faceplate is under vacuum. Performance of the valve is monitored using a fast ion gauge (FIG)[2] which is borrowed from another group laboratory.

Centering of the valve orifice with the beam axis of the mass spectrometer (defined by the line of 3 mm diameter differential pumping holes separating each region after the zeroeth differential) is critical to high ion signals, and this is accomplished using a close-clearance positioning ring in one of two ways. With the zeroeth differential region in place, the ring is precisely mounted on the top plate of this internal chamber, and the valve hangs down from the top of the source chamber, resting inside the ring. Without the zeroeth differential region, the ring is less precisely mounted on a flange bolted to the inside of the source top door, with the valve again hanging down from the top door, resting inside the ring. In this latter case, the internally mounted flange has been carefully aligned, then left in place even when using the zeroeth differential region. Either scheme allows for easy adjustment of the vertical position of the valve under vacuum, because

the tube connecting the valve to the outside is held in place with a Cajon Ultratorr connection, which can be loosened slightly without significant air leakage.

The driver circuit for the pulsed valve was built by Martin Zanni, so no circuit diagram is presented here; the design originates from the Hanna Reisler group at USC. It delivers a voltage pulse to the piezoelectric disk in a "shark's fin" (RC decay) profile, in order not to damage the crystal. The voltage can be adjusted up to 700 V, though typically only 200 V are needed for a well-adjusted poppet. The duration of the high voltage segment can be varied, but is generally kept as short as possible (150? μs) while allowing the valve to fully open. The valve is capable of running at a repetition rate well over 1 kHz, though the optimal rate for producing anions has been found to be 500 Hz. The only modification from earlier incarnations of the design for implementing this high speed is the use of a higher current power supply, and a Bertan 205-01R unit (1 kV, 30 mA) meets these requirements.

1.3.2. Electron gun

The electron gun (Fig. 4 and also Fig. 5) delivers a beam of electrons at 1-2 keV to the gas pulse emerging from the pulsed valve, creating positive ions and slow, secondary electrons which then attach to neutral molecules and cool in the ensuing supersonic expansion. It is a continuous device, unlike most other source components. It has been found that a fairly diffuse beam, aimed 1-3 mm below the pulsed valve orifice, produces the highest levels of anions, though the exact conditions vary considerably with the anion of interest. In certain instances, a pulsed discharge source (see, for instance, Cyr[2]) produces higher anion intensities, but the production of I_2^- and I_2^- clusters, in particular, have only been made successfully with the electron gun.

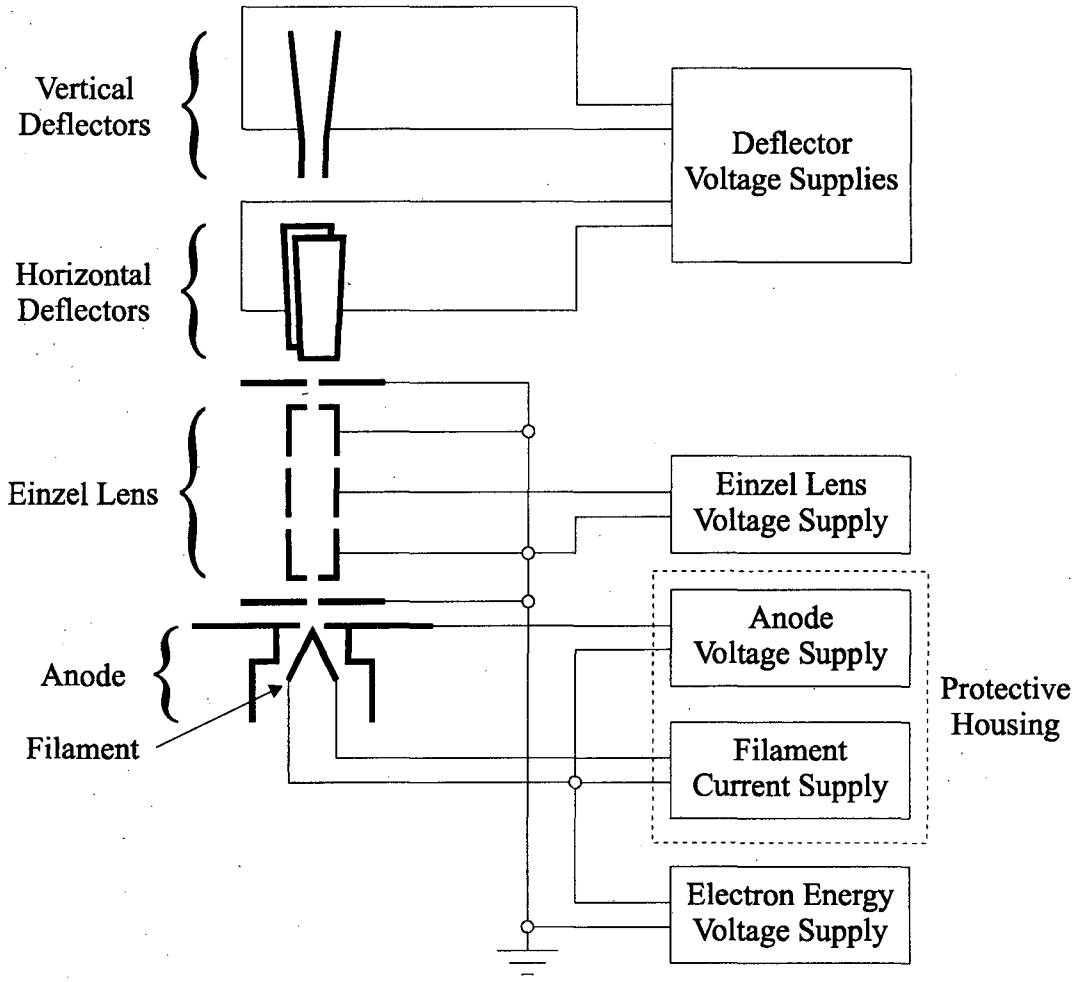


Fig. 4. Electron gun schematic diagram.

The electron gun is modified from a "rejected" Tektronix oscilloscope electron gun to accommodate a custom filament mount and anode cup. Filaments are made of thoriated iridium powder on platinum ribbon, custom ordered from Electron Technologies, Huntingdon Valley, PA. The filament and anode cup are floated at –1 to –2 kV, passing a current of 5-7 A; an anode bias voltage of –30 V is typically used. An Einzel lens, and horizontal and vertical deflectors, are used to optimally position the beam in the gas expansion. Bertan 205-05R high voltage supplies (5 kV, 5 mA) are used

for the float voltage and Einzel lens, while a 150 V Acopian A0150NT05 power supply is used for the anode bias, and an Kepco ATE6-10M current supply for the filament current.

The deflector circuits have been susceptible to large current fluctuations caused by the electron beam nearby to the deflector plates, and as a result, the original circuit, in which a voltage divider was used, had to be abandoned. The new circuit (Fig. 5), based on the anode power supply, uses Acopian A0150NX05 adjustable 150 V power supplies to better deal with the high currents involved in maintaining a deflection voltage.

A copper Faraday cup (Fig. 2), mounted opposite the pulsed valve from the electron gun, is used to measure the electron current, typically 100-500 μA. It is biased to +9 V using a battery attached outside the source chamber. The cup is soldered to a ¼" diameter copper rod, which passes through the top of the source chamber, isolated electrically by an o-ring found in the Cajon Ultratorr connection. The cup can be repositioned in the beam while under vacuum, in the same manner as for the pulsed valve.

1.3.3. Zeroeth differential region

The "zeroeth" differential region, so named since it was built after the first and second differential regions were completed, yet precedes both of them in the differential pumping order, is a removable chamber mounted inside the source. Because of its wholly internal location, it is awkward to measure the pressure, but an ionization gauge is available in the arm of the roughing line. The purpose of the region is to reduce the pressure in the vicinity of the extraction and acceleration plates, which is essential for optimal focusing of the mass spectrometer. The pressure inside this region is typically

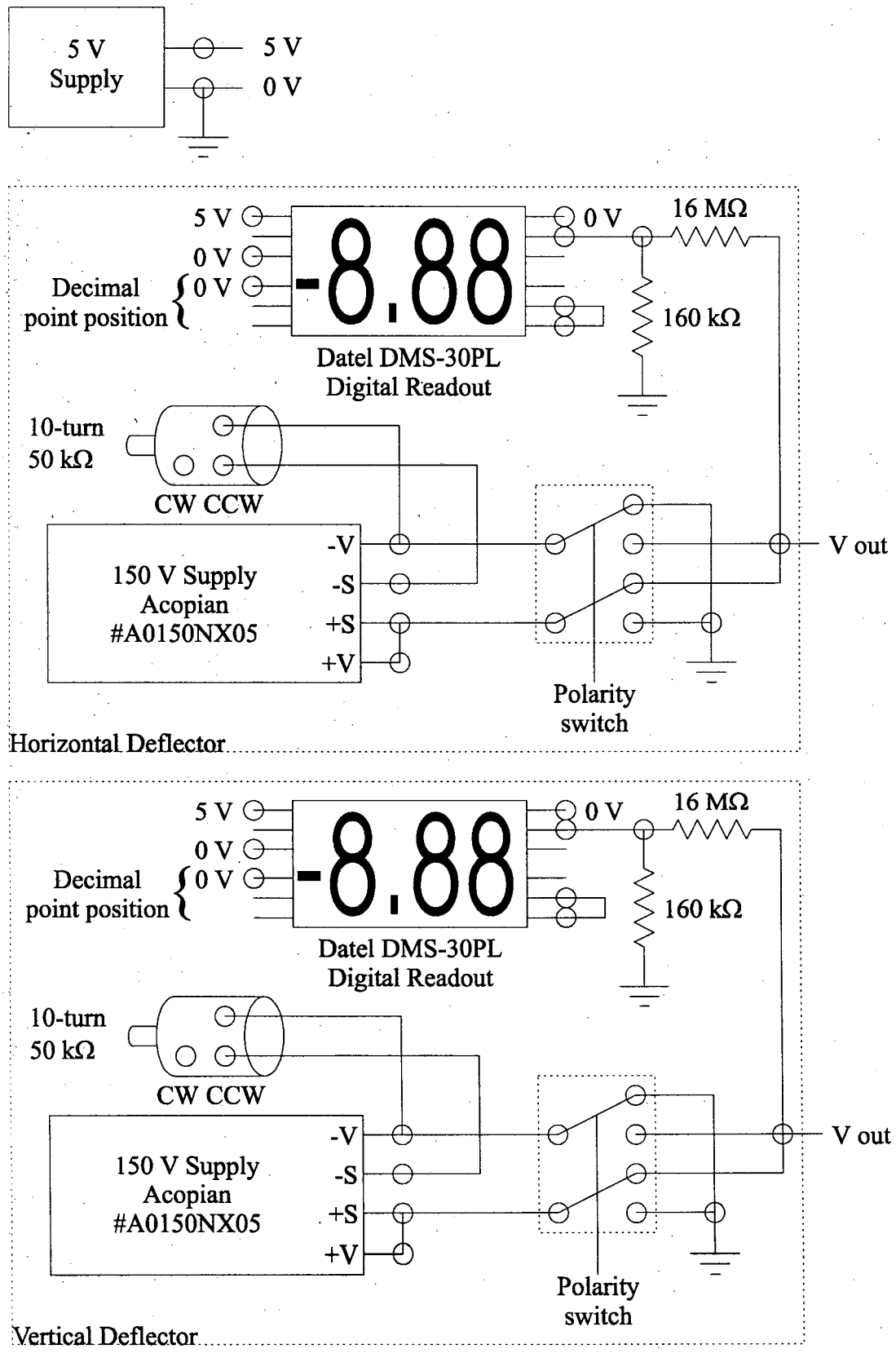


Fig. 5. Electron gun circuit diagram.

~2.5-10 times lower than inside the source region, depending on the distance between the skimmer and pulsed valve. It has been shown to assist in the production of large $I_2^-(Ar)_n$ and $I_2^-(CO_2)_n$ clusters. The drawback of using the region, other than having a more cramped and less accessible source chamber, is that the maximum gas load in the source is reduced by approximately 2×, since only half the pumping capacity is available to evacuate the source region.

The region is used in conjunction with a stainless steel skimmer, which is mounted directly under the pulsed valve and of which several sizes are available. Typical hole diameters used are 2-6 mm, the largest one being the most successful for the production of large I_2^- clusters. The distance between valve orifice and skimmer can be adjusted while under vacuum, as explained above in the description of the pulsed valve. The valve mounting ring atop the chamber aligns the valve precisely with the skimmer.

1.3.4. Extraction and acceleration

Three stainless steel electrode plates (see Fig. 3) located in the zeroeth differential region serve to "extract" the cold anion plume into the time of flight mass spectrometer, which constitutes the rest of the vacuum chamber. (Technically speaking, when the zeroeth differential region is not being used, the extraction and acceleration plates are located in the source region.) A pulsing circuit, built by Martin Zanni and detailed in his dissertation, delivers rapidly falling (100 ns) high voltage pulses to the extraction and acceleration plates simultaneously; the third plate, serving as a partition between the zeroeth and first differential regions, is always grounded. The region between the extraction and acceleration plates is referred to as the extraction region, while the smaller region between the acceleration and ground plates is called the acceleration region. The

voltage on the extraction plate is more negative than the acceleration plate during the pulse, so that anions are accelerated away from the extraction plate, toward the detector region. The maximum voltage on either plate is currently limited to 2 kV, though it is straightforward to increase this limitation by adding extra 1 kV MOSFET stages. Typical voltages used are 1.3 kV for the extraction plate, and 1.0 kV for the acceleration plate.

1.4. First and second differential regions

A diagram of the first and second differential regions is shown in Fig. 6. The first differential region consists of a stainless steel tee mounted on the back side of the source chamber, and a box mounted on the opposite side, extending into the zeroeth differential chamber. Mounted inside the box are a set of horizontal and vertical ion deflector plates, with wires ultimately connecting to feedthroughs on the top of the source chamber. Mounted inside the tee, in the arm away from the source, is the first Einzel lens. A manual gate valve is bolted to the bottom of the tee, with the diffusion pump mounted underneath. The top flange of the tee holds an ionization gauge. The second differential region is separated from the first differential region by a pneumatic gate value. This region contains a set of deflectors on the side nearest the source, and a second Einzel lens on the side opposite. The top flange contains feedthroughs, and an ionization gauge, while the bottom flange holds the turbomolecular pump. A fifth flange is blank, to allow for access to components.

1.4.1. Ion deflectors

Both sets of deflectors are controlled by a circuit designed and built by Martin Zanni, so the diagram will appear in his dissertation. In brief, each circuit consists of a

fixed, 200 V power supply connected to a potentiometer, arranged such that a dynamic range of –50 to +50 V is obtained by turning the potentiometer only; no polarity switch is required. This is an adequate range in virtually all circumstances, so the circuit has worked well. A digital readout is provided for each voltage, to aid in reproducing settings.

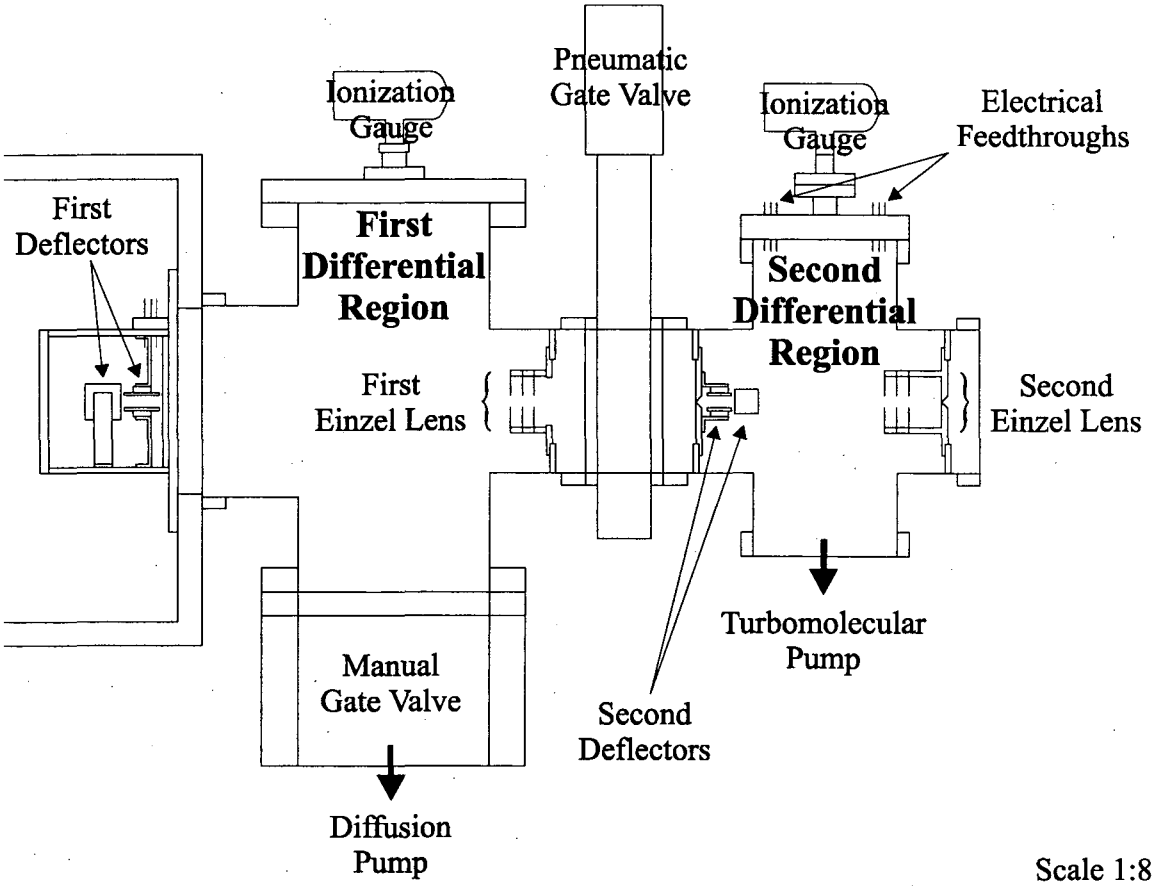


Fig. 6. First and second differential regions.

1.4.2. Einzel lenses

The first Einzel lens is used to focus the ion beam in the plane perpendicular to the travelling direction. A negative voltage of a few hundred volts (Bertan 355) applied to the central plate is required for optimal ion signals. Intensity is fairly sensitive to the voltage; in general, a value of 400-500 V is necessary. The second Einzel lens is not

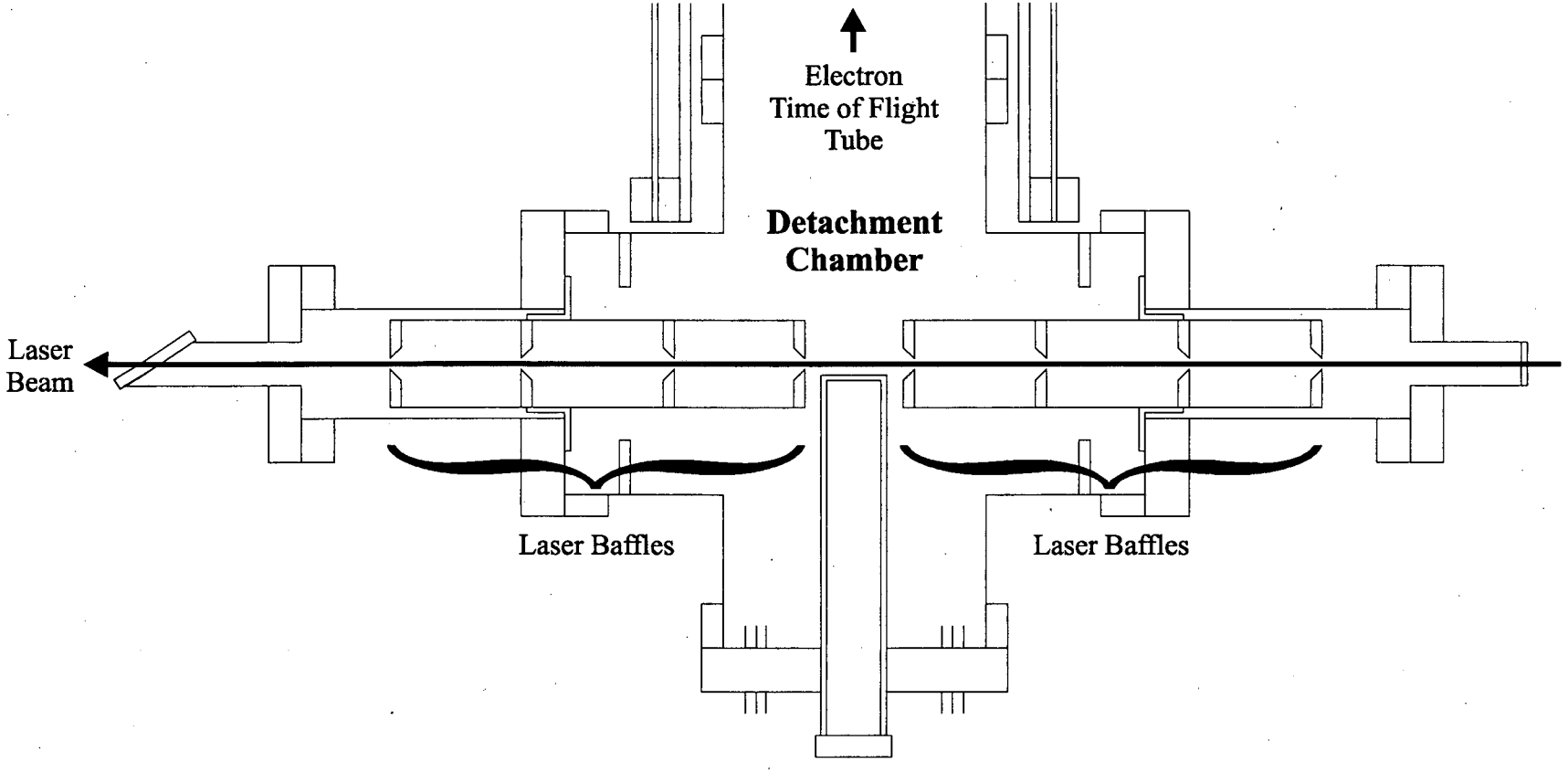
typically used. This is not to say it has no effect on the ion signal; on the contrary, a voltage of several hundred (negative) volts greatly increases the ion signal. However, this improvement also seems to decimate the electron signal (when the laser is present), and the best compromise seems to be to leave the lens at 0 V. This problem is not entirely understood, but appears to stem from the different locations of the laser focal point and the ion detector (about 21.5 cm); with the Einzel lens located only 25.5 cm away from the laser focus, the longitudinal focus can be quite different in the two locations. It is possible that careful optimization of settings may well increase the electron signal above its current performance level.

1.5. Detector region

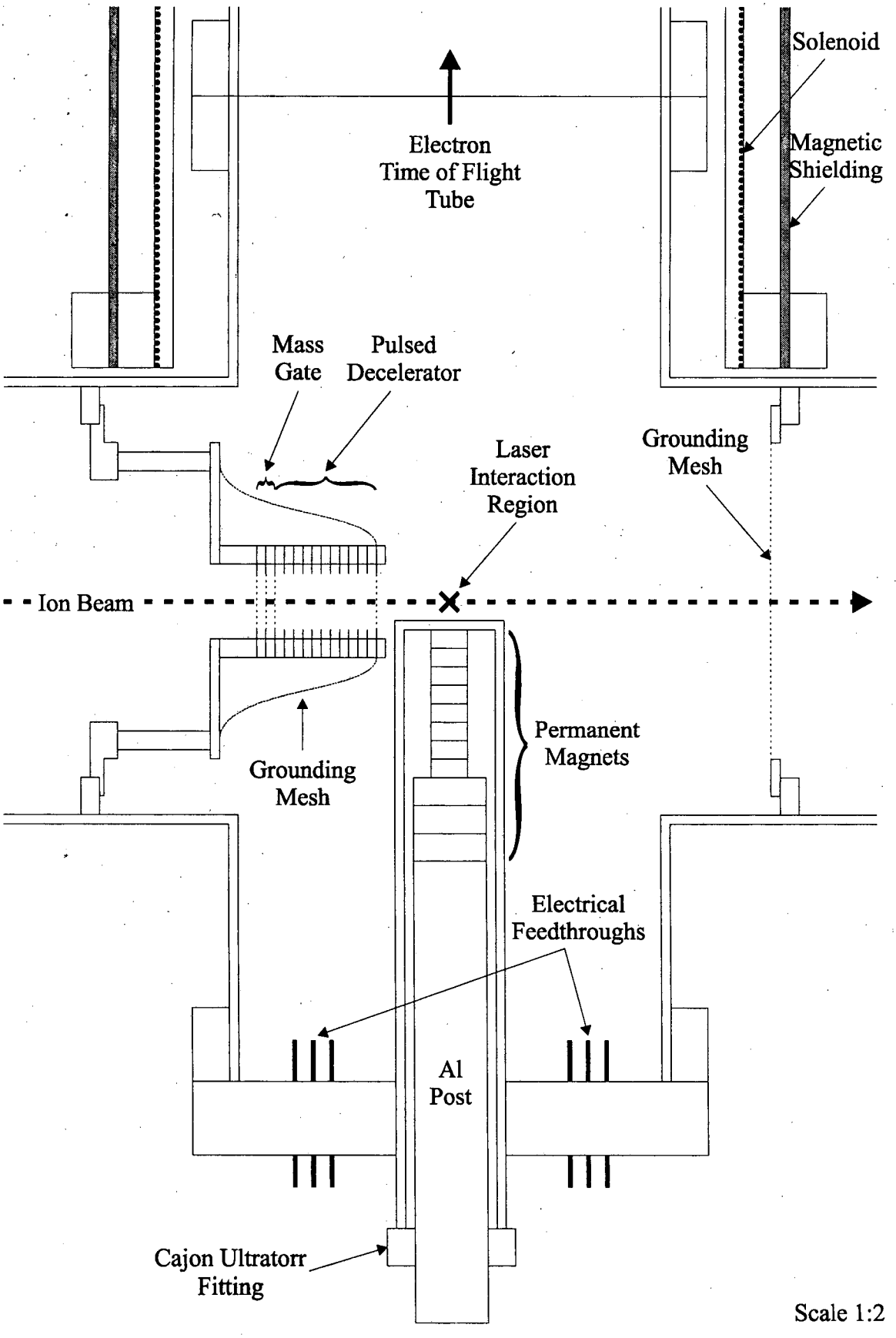
The detector region is the heart of the FPES machine, where ion beams, laser beams, electrons and photofragment ions all share space. It consists of two cross-shaped chambers, and a long tube for electron time of flight. In the "detachment" chamber (Figs. 7 and 8), the left and right flanges each hold a set of laser windows and baffles. Mounted in the front arm of the chamber is a mass gate/ion decelerator assembly, while the rear arm contains a wire grid covering the entire area of the tube, to prevent stray fields from the ion detector located behind the mesh from affecting electron trajectories. The bottom flange contains electrical feedthroughs for the mass gate/ion decelerator, and a hollow post extending close to the midline of the chamber, inside of which (outside vacuum) is a set of removable, strong permanent magnets, constituting the bottom half of the magnetic

Fig. 7. Detachment chamber – front view (on next page).

Fig. 8. Detachment chamber – front view (on page 32).


Electron
Time of Flight
Tube
Detachment
Chamber
Laser
Beam
Laser Baffles
Laser Baffles

Scale 1:4


Solenoid
Magnetic Shielding
Electron Time of Flight Tube
Mass Gate
Pulsed Decelerator
Laser Interaction Region
Grounding Mesh
Ion Beam
Grounding Mesh
Permanent Magnets
Electrical Feedthroughs
Al Post
Cajon Ultratorr Fitting

Scale 1:2

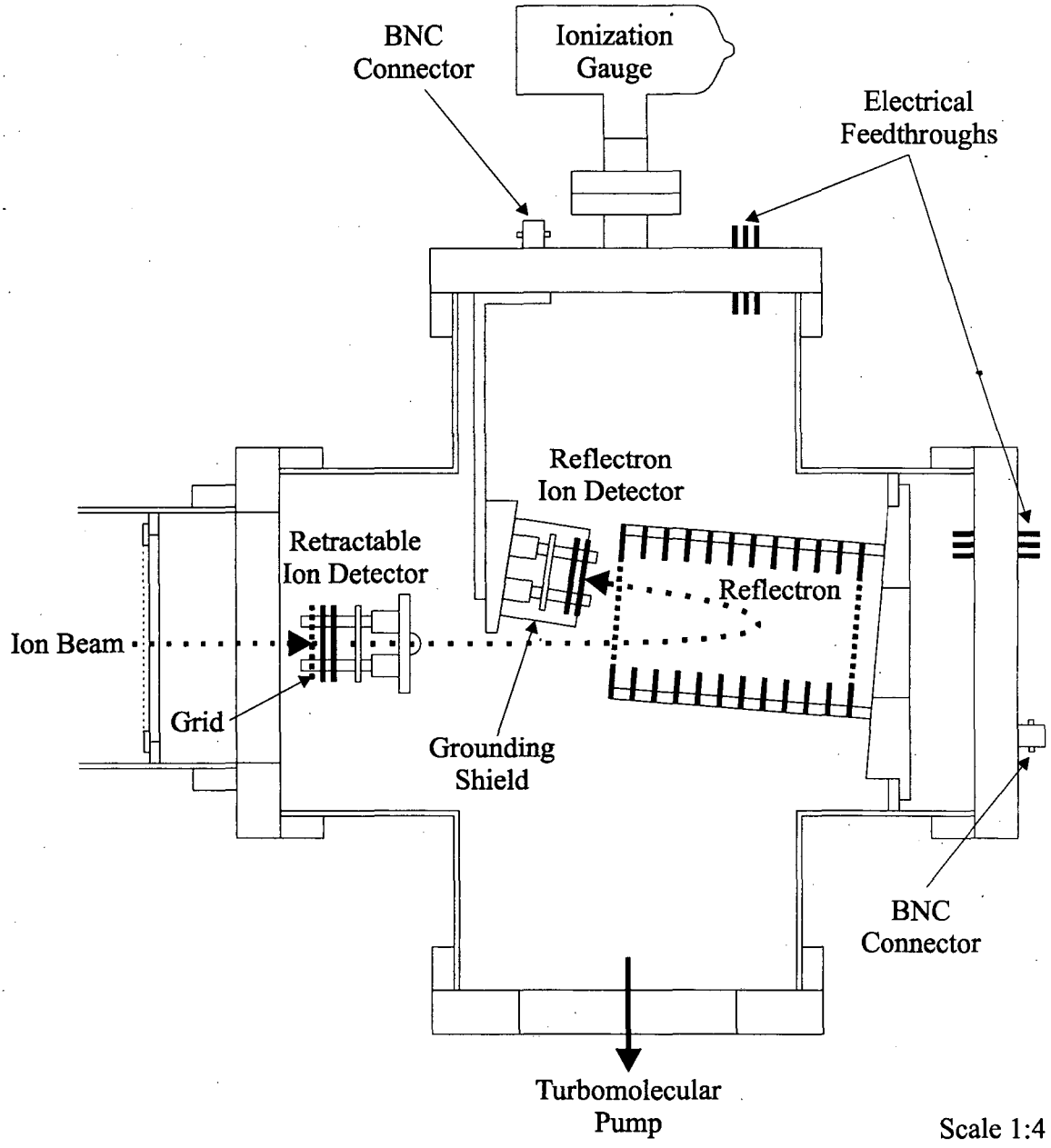


Fig. 9. Reflectron chamber.

bottle. Above this is a 1.17 m long electron flight tube ending at an electron detector. Outside vacuum, the tube is encased in a removable plastic cylinder wound with wire to create a weak solenoid, the top half of the magnetic bottle. Around the solenoid is a single layer of Hypernom magnetic shielding. The "reflectron" chamber (Fig. 9) holds a turbomolecular pump on the bottom flange, reflectron on the rear flange, and reflectron

ion detector, electrical feedthroughs, and ionization gauge sharing the top flange. A retractable ion detector is mounted on a small left flange, along with associated feedthroughs.

1.5.1. Laser windows and baffles

Suprasil windows, transparent to ultraviolet light down to ~190 nm, are mounted on the ends of narrow tubes extending from the left and right arms of the detachment chamber (Fig. 7). Although the left (exiting) window is still mounted at Brewster's angle to minimize reflections from vertically polarized light, the right (entrance) window has been replaced with a 2 mm thick window mounted nearly normal to incident light. This was done primarily to reduce the thickness of glass encountered by the laser as much as possible. It also gave flexibility if horizontally polarized light was desired, though the reflection of oppositely polarized light from a Brewster's window is only ~15%. The reason for using a thinner window stem from group velocity dispersion (GVD) or broadening considerations of femtosecond ultraviolet pulses, which are discussed in section 2.1.

A baffle tube is mounted inside each arm. The tube is black anodized aluminum containing a set of three baffle disks, also black anodized aluminum, with a ¼" hole drilled through the center, one side chamfered to form a knife edge at the rim. The disks in the tube nearest the laser beam, as well as the two innermost disks in the tube away from the beam, are oriented with the knife edge pointing toward the laser (chamfer in back); the remaining two disks in the far tube are oriented pointing away from the laser (chamfer in front). The purpose of the baffles is to reduce scattered light from the edges of the laser beam, which may be blotchy, or overall defocused. It also aids in aligning the

laser beam, as a narrow range of input angles can actually get through to the other side. The main method of laser alignment relies, however, on a pair of reference pinholes separated by several meters, located on either side of the vacuum chamber.

1.5.2. Retractable ion detector

All three charged-particle (ion, reflectron ion, and electron) detectors utilize a pair of microchannel plates held at high positive voltage, which, when struck by an anion or electron, generate an electron cascade whose current can be measured on an ordinary oscilloscope. These high performance devices require a 1-2 kV potential, producing signals with a rise time of ~1 ns.

The ion detector (Fig. 10) consists of a front wire mesh (grid) held by a metal sandwich; the microchannel plate pair (Galileo 1390-2500, 25 mm dia.) consisting of front and rear mounting rings, and a thin loop inserted between the plates; and the anode. The entire assembly is connected together via three sets of insulating vespel spacers compressed together with loose springs, and bolted to small metal cylinders; the cylinders are then held in place on a mounting plate with set screws. The voltage of the grid can be switched negative to repel ions and detect fast-moving neutral molecules, but it is otherwise kept at 0 V. Each of the remaining four components is connected outside the vacuum chamber by a resistor chain, with the front plate held at 0 V, and the anode held at +1.5 kV. In order to measure signals at a safe DC voltage, the anode is capacitively coupled inside the vacuum chamber to a separate BNC signal connector.

When the machine was first built, the ion detector was mounted on the back flange, behind the reflectron, but this location was discovered to be too far from the laser interaction region (55 cm) to allow for optimal ion focusing. Therefore, the entire

detector has been designed to retract via ultrahigh vacuum baffles-sealed translator arm (MDC Corp. SBLM-133-4), which moves the assembly entirely out of the way of the ion beam so that reflectron experiments can be performed, but otherwise allows the detector to sit much closer to the laser beam, 21.5 cm.

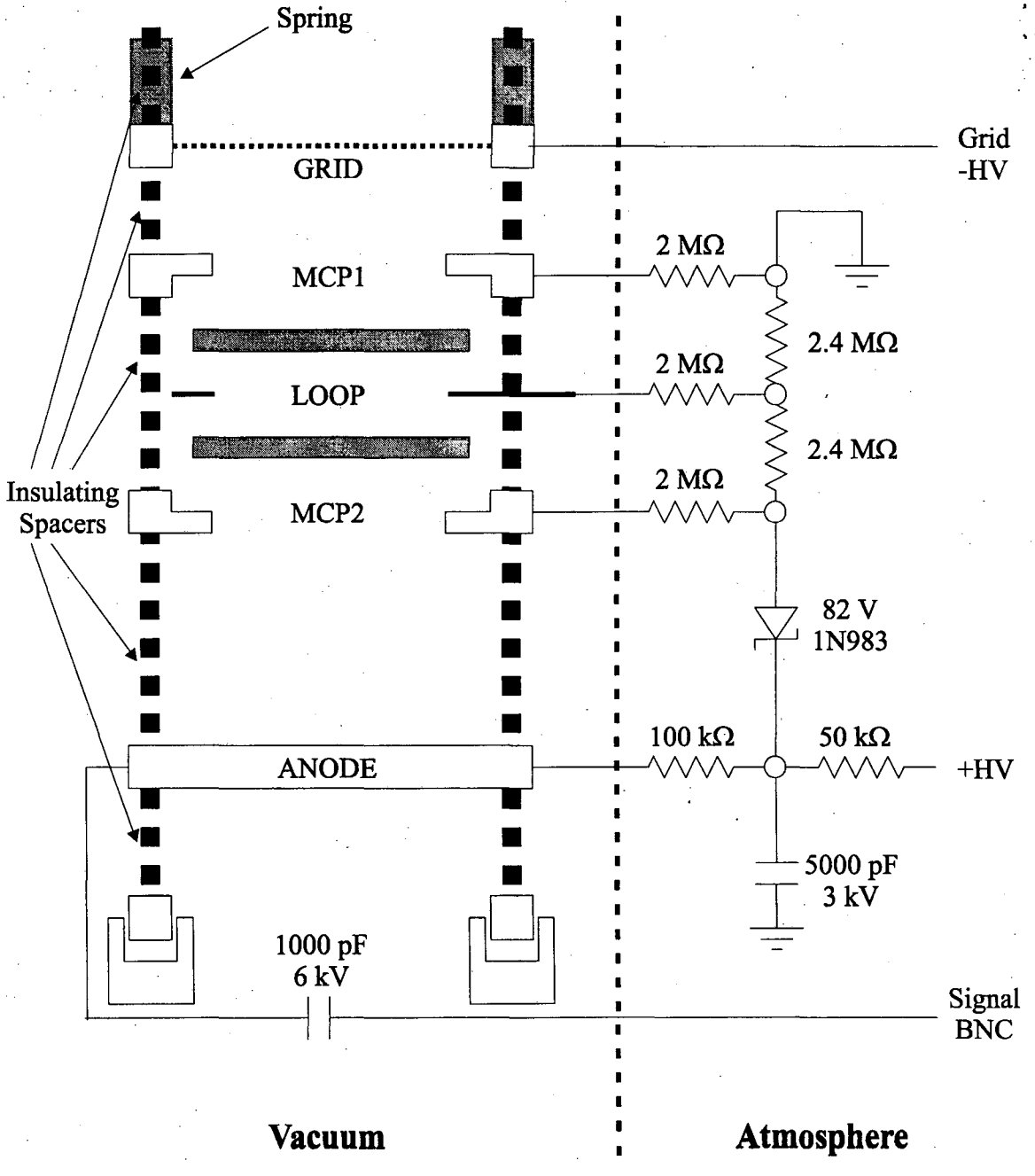


Fig. 10. Ion detector schematic diagram.

1.5.3. Magnetic bottle

The magnetic bottle[4,5] is designed to collect a large (>50%) fraction of photoelectrons while preserving electron energy resolution as much as possible. This is accomplished using two elements: a stack of strong permanent magnets (3000 G surface field) located 9.5 mm below the ion beam axis, and a weak solenoid (20 G) beginning approximately 7.6 cm above the beam axis and continuing for 1.17 m, past the electron detector. The field at the laser interaction point is difficult to measure precisely, but is estimated to be ~1000 G. Electrons in a magnetic field will precess about the field lines, so that as the field changes, the radius of the orbit will increase or decrease accordingly, along with a change in the orbital velocity. Since total energy must be conserved, the forward velocity of electrons headed toward the increasing magnetic field will eventually become zero, resulting in reflection (the so-called "magnetic mirror" effect).[6] All electrons travelling along field lines which intersect the electron detector are therefore collected.

The permanent magnets are a samarium type (Edmund Scientific) in two cylindrical shapes: 1" dia. × 0.38" thick (part # 30963), and 0.5" dia. × 0.20" thick (part # 52861). The arrangement producing the highest electron yield appears to be when they are stacked on top of one another, the 8 small ones on top of the 3 large ones. However, detection efficiency is not critically dependent on this set-up. The stack is glued to an aluminum shaft and the entire post then secured inside the hollow tube with a Cajon Ultratorr connector. Nonmagnetic stainless steel is used in the construction of all parts of the ultrahigh vacuum region. The solenoid consists of a removable plastic tube which is placed around the electron flight tube. One layer of Hypernom magnetic shielding

surrounds the solenoid, to reduce extraneous magnetic fields. Coated copper magnet wire (14 gauge) is wound at 10 turns per inch (3.94 turns per cm) for the entire length of the tube. A Kepco high current power supply (ATE25-4M) maintains an adjustable current of up to 4 A. The magnetic field B at the center of the solenoid can be calculated by:

$$B = \mu_0 nI \tag{1}$$

where μ_0 is the magnetic permeability ($4\pi\times10^{-7}$ H m^{-1}), n is the coiling density (turns/unit length), and I is the current. Thus, a current of 4 A produces a field of 20 G. The choice of wire thickness was based on the maximum voltage of the power supply (25 V). We find, for a ~350 m length of wire, a resistance of 4.5 Ω, which produces a voltage of 18 V at 4 A, well within these limits. The heat dissipated, 72 W, is not found to be significant over the length of the solenoid.

Although it is not trivial to calculate the field surrounding a permanent magnet, such calculations have been done for similar magnetic bottles using an electromagnet (solenoid) in place of the permanent magnet.[5] The combined magnetic field of the strong and weak solenoid fields is found to be highly divergent in the vicinity of the ion beam, the key to the magnetic focusing property. Electrons produced at this "focal point" through photodetachment are repelled by the converging field near the strong magnet, and orient themselves along lines of the field as they travel toward the electron detector.

1.5.4. Electron detector

The electron detector (Fig. 11) uses a pair of large, 75 mm diameter microchannel plates (Galileo 1396-7500) to increase the electron collection area. They are mounted on the top flange of the electron flight tube, behind a wire mesh which is directly bolted to

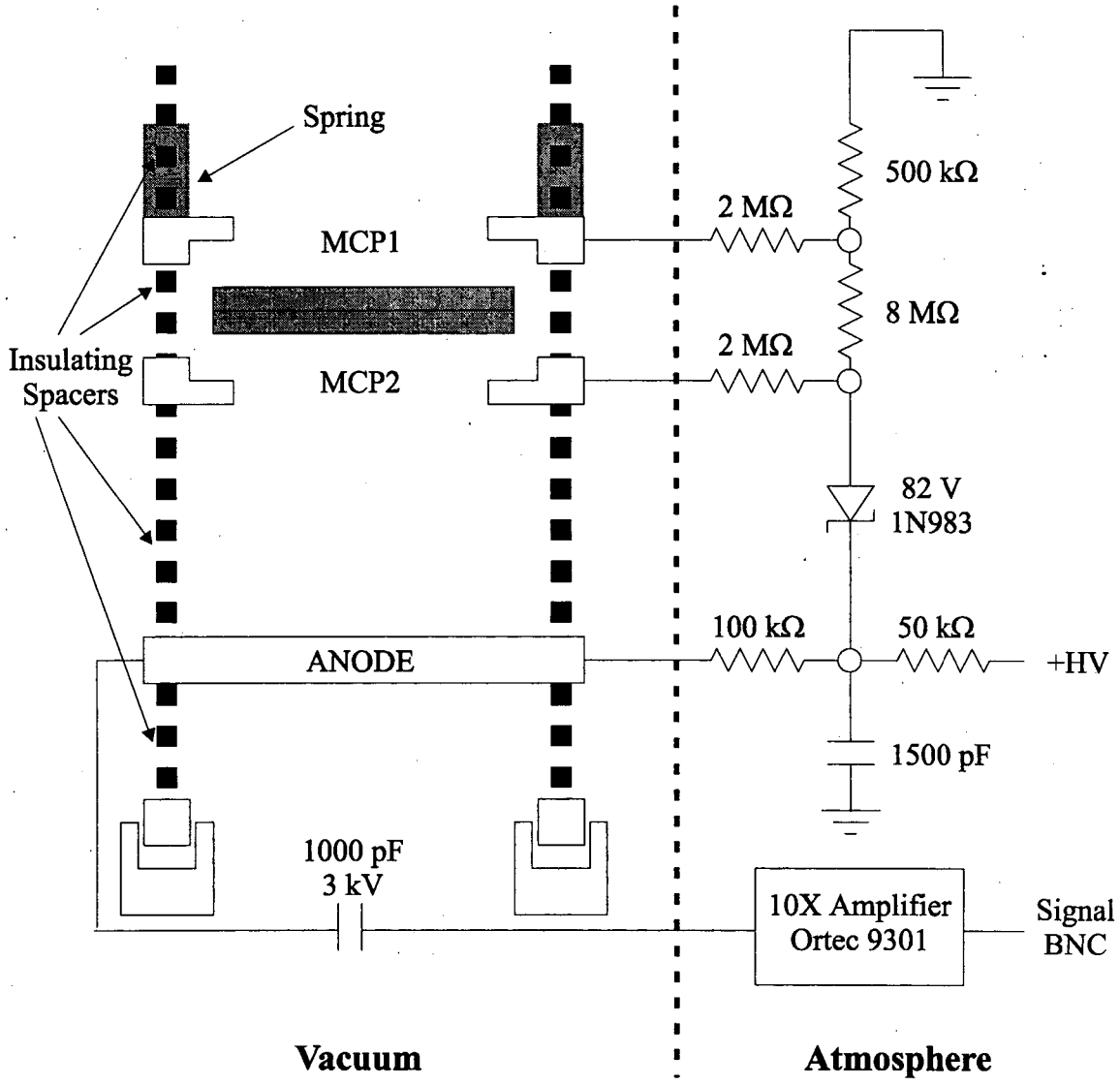


Fig. 11. Electron detector schematic diagram.

the sides of the flight tube, allowing electrons below the mesh to experience an (electrically) field-free region. The microchannel plates are held together via front and back mounting rings only; the middle loop is absent since the plate resistances have been matched, allowing a single voltage to be applied across the pair. Behind the plates is an anode. The assembly is connected together using vespel spacers and loose springs, as for the ion detector. The three components are connected outside the vacuum chamber by a

resistor chain, with the front plate at 0 V, and the anode at –2.2 kV. The anode is capacitively coupled inside the vacuum chamber to a BNC signal connector. A 10× fast preamplifier (Ortec 9301) is used just outside the vacuum chamber to boost the signal prior to digitization.

1.5.5. Mass gate

The mass gate is used to admit a narrow time slice of ions through to the rest of the spectrometer, which is necessary for both the pulsed ion decelerator and reflectron. It consists of three stainless steel plates, each with a grid of fine wires to ensure uniform electric fields, mounted on the same assembly as the ion decelerator (Fig. 8). Normally, the middle plate is held at high negative voltage to repel all ions, but when the ion packet of interest approaches the first plate, the potential is quickly and temporarily dropped to ground to allow admission. After the packet clears the third plate, the potential returns to high voltage.

The success of such a gate relies heavily on a pulsing circuit capable of delivering rising and falling edges of ~100 ns (see dissertation of Martin Zanni). It also requires the close spacing of plates. If the distance between plates is large, a greater amount of time must be spent with the gate "open" (at ground potential), with the consequence of lower mass selectivity. For an assumed mass resolution of 300 and a time of flight distance of 1.3 m, the spatial extent of an ion packet at the detector is 4.3 mm; it will be slightly larger at the mass gate. The distance between the first and last plate of the mass gate, 6.2 mm, is comparable. For a typical flight time of 50 μs, the ions spend 240 ns inside the gate, within the capabilities of the pulsing circuit.

1.5.6. Pulsed ion decelerator

The decelerator consists of a stack of 11 plates spaced 1/8" (3.18 mm) apart, with grids on the first and last plates, and a resistor (1.5 kΩ) joining each plate together; see Fig. 8. The first plate (closest to the source region) is also a part of the mass gate, being normally grounded. The last plate (closest to the ion detector) is permanently grounded. Once the ions have passed the first plate, it is pulsed to high positive voltage, and it is grounded again before the ions pass the last plate. The response time of the plates, ~150 ns, is slightly slower than that of the driving circuit, owing to some capacitance of the stack, but as 1 μs or more is typically needed to decelerate the ions with a 2 kV potential, this is adequately fast. An example of the deceleration capabilities is presented in the section on resolution of photoelectron spectra, described below.

1.5.7. Reflectron

The reflectron (Fig. 9) consists of 12 rings, 4" in diameter with a 2.5" diameter hole, separated by ½" long insulating Vespel spacers. Each plate is connected electrically by a 2 MΩ vacuum-compatible resistor (K&M Electronics, CR1243G2M00G3), with the front plate grounded, and the rear plate held at a high negative voltage. The front and rear plates contain mesh grids to ensure a uniform electric field across their entire surfaces. Anion photofragments entering the reflectron experience a retarding electric field which ultimately reflects them back out. The reflectron is tilted upward by 4.5°, redirecting the path of the ions toward the detector located above the beam axis.

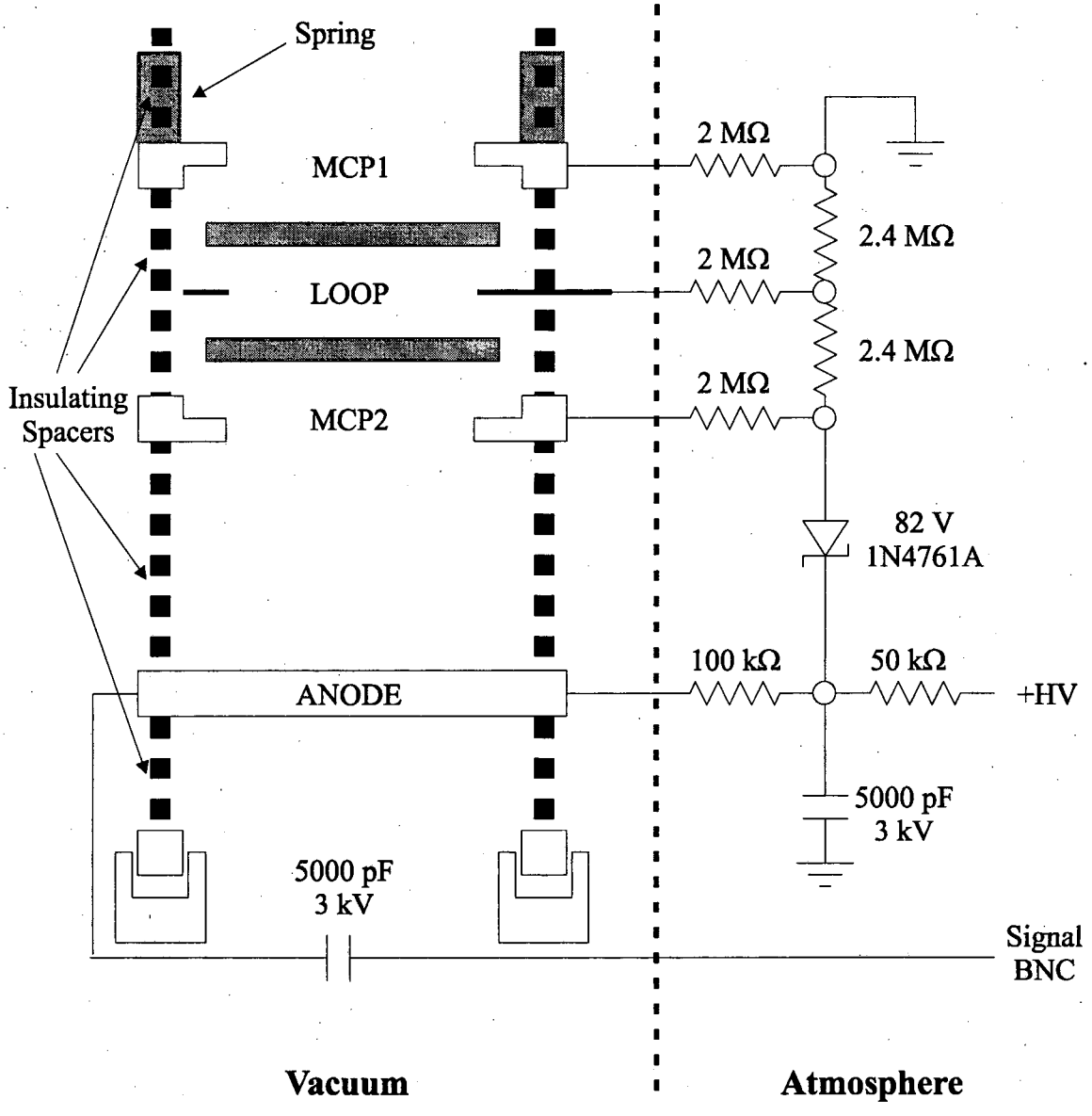


Fig. 12. Reflectron detector schematic diagram.

The reflectron ion detector (Fig. 12) is similar in design to the retractable ion detector. It consists of a pair of 25 mm dia. microchannel plates (Galileo 1390-2500) held together by front and rear mounting rings, with a thin loop inserted between the plates; behind this assembly is the anode. Components are connected together with three sets of insulating Vespel spacers compressed by loose springs, and bolted to small metal cylinders; the cylinders are then held in place on a mounting plate with set screws.

Outside the vacuum chamber is a resistor chain connecting the plates electrically. The front MCP is grounded, while the anode is held at –1.5 kV. The anode is capacitively coupled inside the vacuum chamber to a separate BNC signal connector for measuring signals. The entire assembly is housed inside a grounded cylinder to electrically shield it from the anion beam, which passes underneath it only ¼" away. The detector is tilted downward by 9°, so that the detector face is normal to the reflected photofragment beam.

1.6. Timing

The FPES experiment is pulsed. Anions are created, cooled, extracted, focused, excited and detached by laser beams, and electrons detected, at up to 1 kHz. Therefore, the careful control of different timing elements in the system is essential for a successful experiment.

Because the femtosecond laser is passively modelocked, it cannot be triggered by an external signal, but must instead be used to trigger other parts of the experiment. The NJA-5 oscillator triggers the Clark-MXR DT-505, which drives the Pockels cell at a greatly-reduced repetition rate (500 Hz). The Sync 3 output from the DT-505 serves as the primary trigger for the molecular beam segment of the experiment. From this trigger, two Stanford Research Systems DG535 delay generators ("Stanford boxes") are used to coordinate several devices: pulsed valve driver, extraction and acceleration circuit, pulsed discharge circuit, mass gate circuit, pulsed deceleration circuit, New Focus optical chopper, ion oscilloscope, and an analog-to-digital (A/D) converter for recording autocorrelation and cross-correlation signals. The multichannel scalar (MCS) for collecting electron time of flight spectra is separately triggered by the DT-505 (Sync 2), which is almost coincident (few ns delay) with the arrival of the laser pulses in the

detector region. Although needs change slightly depending on the task at hand, an overall schematic diagram illustrates the general timing approach of the experiment (Fig. 13).

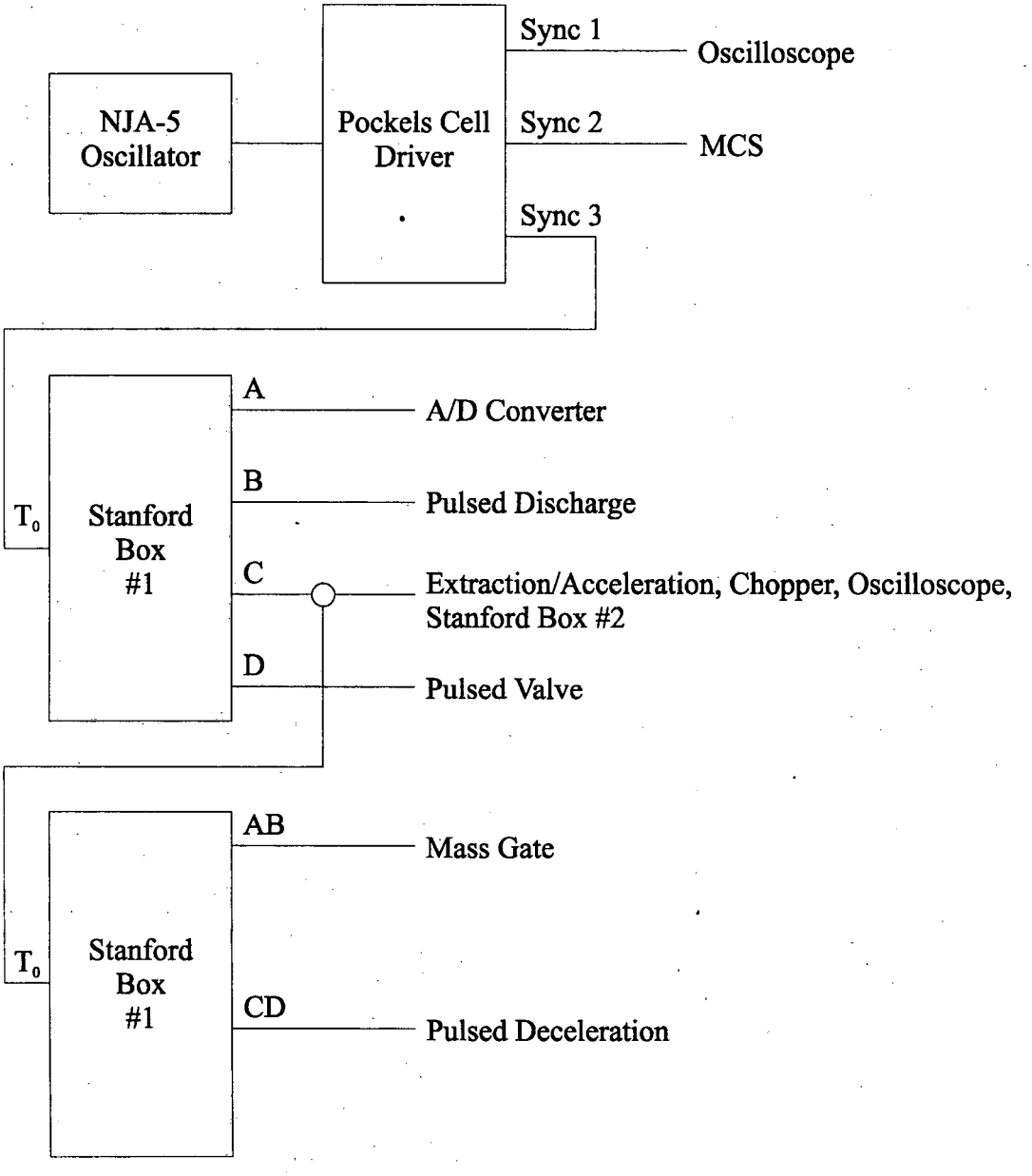


Fig. 13. Timing signals of the FPES experiment.

1.7. Ion time of flight

The spacing of plates in the extraction and acceleration regions, and the electric fields in each region, critically affect the focusing of the spectrometer, which is

monitored by the sharpness of features at the ion detector. Wiley and McLaren[1] first derived the equations needed to determine these parameters for an idealized mass spectrometer, assuming negligible initial kinetic energy to the ions, where D is the distance between the ground plate and the ion detector:

$$D = 2s_0 k_0^{\frac{1}{2}} \left(1 - \frac{1}{k_0 + k_0^{\frac{1}{2}}} \frac{d}{s_0} \right) \tag{2}$$

$$k_0 = \frac{s_0 E_s + dE_d}{s_0 E_s}. \tag{3}$$

Here s_0 and d are the distances the ions travel in the extraction and acceleration regions, and E_s and E_d are the electric fields in each of these regions. Since the pulsed valve is centered over the extraction region, the width of this region is $2s_0$, and s_0 only represents an average distance traveled by the ions. Note that D is independent of ion mass, which is a key advantage of the design. Using typical parameters for the machine ($2s_0$ = 6.35 cm, d = 2.54 cm, E_s = 50 V/cm, E_d = 400 V/cm), one obtains a focal point of 1.18 m, close to the measured distance of 1.3 m. However, in reality it has been found that the anion spectrum is slightly dependent on mass, requiring a larger extraction field to focus heavier anions.

The overall resolution $M_s = m/\Delta m$ of the spectrometer depends on the spread of ion energies, which is in turn determined by the spatial spread Δs of the ions in the extraction region:

$$M_s = 16 k_0 \left(\frac{s_0}{\Delta s} \right)^2. \tag{4}$$

Without a skimmer, Δs may be as large as $2s_0$, the width of the extraction region, giving M_s = 30 in the current setup. However, ions are typically not distributed uniformly across

the extraction region, and we find $M_s \approx 200$ under typical operating conditions, implying $\Delta s \approx 2.5$ cm. With a skimmer, Δs may be smaller, producing better resolution ($M_s \approx 400$).

Ions of mass m arrive at the detector at time T:

$$T = 1.02\text{cm}^{-1}\text{amu}^{-\frac{1}{2}}\text{eV}^{\frac{1}{2}}\mu\text{s}\sqrt{\frac{m}{2U_{ion}}}\left(2k_0^{\frac{1}{2}}s_0 + \frac{2k_0^{\frac{1}{2}}}{k_0^{\frac{1}{2}}+1}d + D\right) \tag{5}$$

$$U_{ion} = qs_0E_s + qdE_d \tag{6}$$

where q is the charge on the ion, and U_{ion} is the total ion kinetic energy.

Because distances are not known precisely, a mass spectrum is generally calibrated from the measured arrival times of two known masses using the simpler equation

$$m = S(T - t_0)^2 \tag{7}$$

where S and t_0 are empirically determined slope and intercept parameters. A sample mass spectrum is shown in Fig. 14(a). The spectrum was obtained from an I_2/Ar/CO_2 expansion mixture, and calibrated from two known points, I_2^- and $I_2^-(CO_2)_{16}$. The dominant masses are $I_2^-(CO_2)_n$ and $(CO_2)_n^-$ clusters, as well as $I_2^-(CO_2)_n(H_2O)$ and $(CO_2)_n(H_2O)^-$ in smaller amounts. Fig. 14(b) shows an expanded view in the vicinity of $I_2^-(CO_2)_{16}$, where the various cluster progressions can be seen clearly. As this spectrum was focused for $I_2^-(CO_2)_{16}$, the mass resolution is highest in this region, about 370.

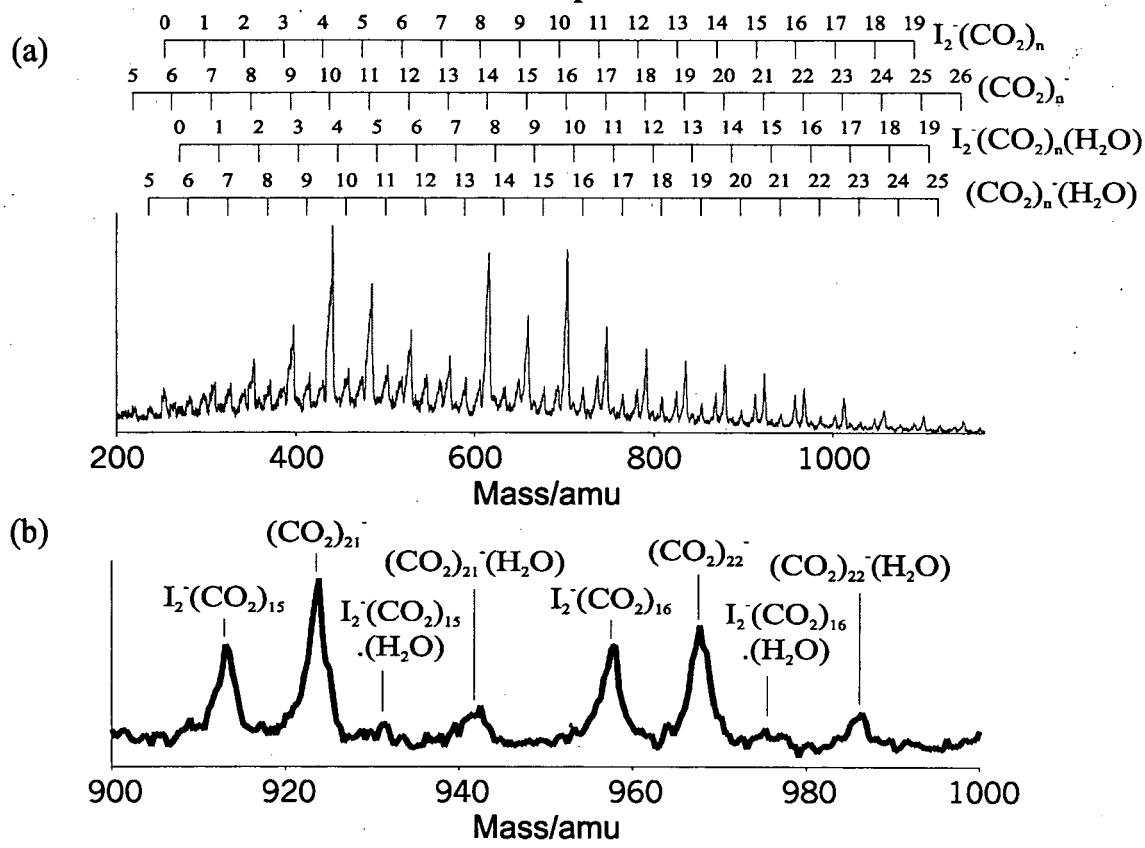


Fig. 14. Sample mass spectrum for an $I_2/Ar/CO_2$ expansion. (a) Full spectrum. (b) Expanded view near 958 amu [$I_2^-(CO_2)_{16}$], showing cluster progressions.

1.8. Electron time of flight

Electrons photodetached at the laser interaction region initially travel in all directions, but the magnetic bottle forces them to turn toward the electron detector, where they travel with constant electron kinetic energy (U_e) which can be measured by the time of flight technique:

$$U_e = \frac{m_e v_e^2}{2} = \frac{m_e L^2}{2(t - t_0)^2} \tag{8}$$

where m_e is the electron's mass, v_e is its absolute velocity, t is its arrival time, t_0 is the laser firing time, and L is the length of the flight tube. The quantity $t - t_0$ is the time of flight. In general, t_0 and L are empirical quantities which must be determined from

calibration. Typically, I^- is used for this purpose; it displays two atomic transitions ($I\ ^2P_{3/2} \leftarrow I^-\ ^1S_0$ and $I\ ^2P_{1/2} \leftarrow I^-\ ^1S_0$) split by a roughly 1 eV spin-orbit energy. Using the third harmonic frequency of the laser (260 nm), the electron kinetic energies of these transitions are 1.709 and 0.766 eV, respectively.

1.8.1. Resolution

Several factors affect the resolution of the electron spectrometer. The most significant is ion velocity v_{ion}, which is added as a vector to the center-of-mass electron velocity v_e^{cm}, giving the apparent electron velocity in the laboratory frame v_e:

$$\vec{v}_e = \vec{v}_e^{\,cm} + \vec{v}_{ion}. \tag{9}$$

The largest difference in velocities is twice the ion velocity: $\Delta v_e = 2v_{ion}$. The resulting spread in the laboratory frame electron kinetic energy ΔU_e is:[5]

$$\Delta U_e = m_e v_e^{cm} \Delta v_e = 2 m_e v_e^{cm} v_{ion} = 4\sqrt{\frac{m_e U_e^{cm} U_{ion}}{m_{ion}}} \tag{10}$$

where U_{ion}, U_e and U_e^{cm} are the kinetic energies of the ion, laboratory frame electron and center-of-mass frame electron, respectively, and m_{ion} is the mass of the ion. Typical parameters are U_e^{cm} = 1 eV, U_{ion} = 1200 eV and m_{ion} = 254 amu (I_2^-), giving $\Delta U_e \approx 200$ meV. However, this spread can be reduced significantly by decelerating the ion beam; this will be dealt with in the following section.

Since the apparent electron kinetic energy is changed so significantly by the ion velocity effect, an instrument reponse function was derived, for an isotropic electron angular distribution. The probability p of finding an electron with lab energy U_e, based on a center-of-mass energy U_e^{cm}, is:

$$p(U_e, U_e^{cm}) = \sqrt{1 - \frac{m_{ion}}{4m_e U_e U_e^{cm}}\left(U_e - U_e^{cm} - \frac{m_e U_{ion}}{m_{ion}}\right)^2} \quad (11)$$

Note that the offset, $m_e U_e / m_{ion}$, is generally very small (2.6 meV using the above parameters) in comparison to $U_e - U_e^{cm}$.

Another source of resolution loss is the finite time required for electrons ejected in a direction away from the electron detector to be reflected by the rapidly converging magnetic field lines near the permanent magnet. The time spread Δt is proportional to $1/v_e$ or $1/U_e^{1/2}$. Since from Eq. 8, $U_e \propto 1/t^2$, we find that

$$\Delta U_e = -\frac{2U_e \Delta t}{t} \propto U_e. \quad (12)$$

Thus, the relative uncertainty $\Delta U_e / U_e$ is independent of energy. It is difficult to calculate Δt exactly, but in a similar magnetic bottle spectrometer which used a second solenoid in place of the permanent magnet, the magnetic fields were known exactly, and trajectory calculations were performed.[5] They obtained $\Delta t = 45$ ns $eV^{1/2}/U_e^{1/2}$ which gives $\Delta U_e / U_e \approx$ 4% using our flight length.

After reflection by the magnetic bottle, electrons do not point precisely toward the detector but precess with a small angle about the magnetic field axis of the solenoid. The maximum value of this angle Θ_{max} depends on the ratio of initial to final magnetic fields, referred to as the "degree of parallelization":[5]

$$\Theta_{max} = \sin^{-1}\sqrt{\frac{B_f}{B_i}} \quad (13)$$

where B_i and B_f are the initial and final magnetic fields, respectively. The effect of Θ_{max} on resolution is to retard the arrival of an electron, resulting in an energy uncertainty of

$$\frac{\Delta U_e}{U_e} = \frac{B_f}{B_i}. \tag{14}$$

The field at the laser interaction point, ~1.0 cm above the top magnet, is not known precisely, but is estimated to be ~1000 G. The final field B_f used is 20 G or less. Thus, $\Delta U_e/U_e \leq$ ~2%.

The factor which most limits resolution in more traditional photoelectron spectrometers is the uncertainty in timing arising from the finite pulse duration of ns lasers, the interaction volume of the ions, and changes in flight length due to differences in beam position. The minimum time duration of the digitizing electronics is generally not an issue. For the FPES machine, the laser pulse duration is obviously not a limiting factor, but the other two items must still be considered. The size of the interaction volume of laser and ion beam is on the order of 3 mm, the differential hole diameter. Changes in laser beam position are probably smaller than 3 mm. The energy uncertainty is:[7]

$$\Delta U_e = \frac{2U_e \Delta L}{L}. \tag{15}$$

Thus, with ΔL = 3 mm and L = 1.3 m, $\Delta U_e/U_e$ = 0.5%, a very small effect.

1.8.2. Pulsed ion deceleration

In order to improve electron kinetic energy resolution, it is necessary to reduce the ion velocity and therefore the electron velocity offsets. Simply running at a lower ion beam energy is not feasible, since these energies are required for focusing in a reasonable flight distance, minimizing the importance of intrinsic kinetic energy of the beam, minimizing the effects of stray fields, and efficient ion detection. Therefore, the ions must be decelerated just prior to interaction with the laser beams.

The most straightforward scheme for accomplishing this would, at first glance, seem to require only a static retarding electric field prior to laser photodetachment, with appropriate rereferencing of the beam either before or after deceleration. This was the approach taken in some of the first magnetic bottle photoelectron spectrometers.[5] However, the kinetic energy spread of the ions inherent in the Wiley-McLaren time of flight scheme, often 10% or more of the total energy, makes this approach impractical if significant signal levels are needed, because the energy spread is the same before and after deceleration:

$$U_i = U_{ion} + q\Delta sE_s \tag{16}$$

$$U_f = U_i - qxE_x \tag{17}$$

$$\Delta U_i = \Delta U_f = q\Delta sE_s \tag{18}$$

where U_i and U_f are the initial and final ion kinetic energies, respectively, x is the length of the deceleration region, and E_x is the deceleration field.

A better deceleration technique, known as "impulse" or "momentum" deceleration, employs a retarding electric field applied to the ion packet for a finite amount of time t_x while inside the deceleration region:

$$v_f = v_i - \frac{qE_x t_x}{m_{\text{ion}}} \tag{19}$$

$$\Delta v_f = \Delta v_i = \frac{v_i \Delta U_i}{2U_i} \tag{20}$$

$$\Delta U_f = \frac{2U_f \Delta v_f}{v_f} = \frac{v_f}{v_i}\Delta U_i = \sqrt{\frac{U_f}{U_i}}\Delta U_i \tag{21}$$

where v_i and v_f are the initial and final ion velocities. Thus, the spread in velocities, rather than energies, is preserved, resulting in a much smaller final energy spread ΔU_f.

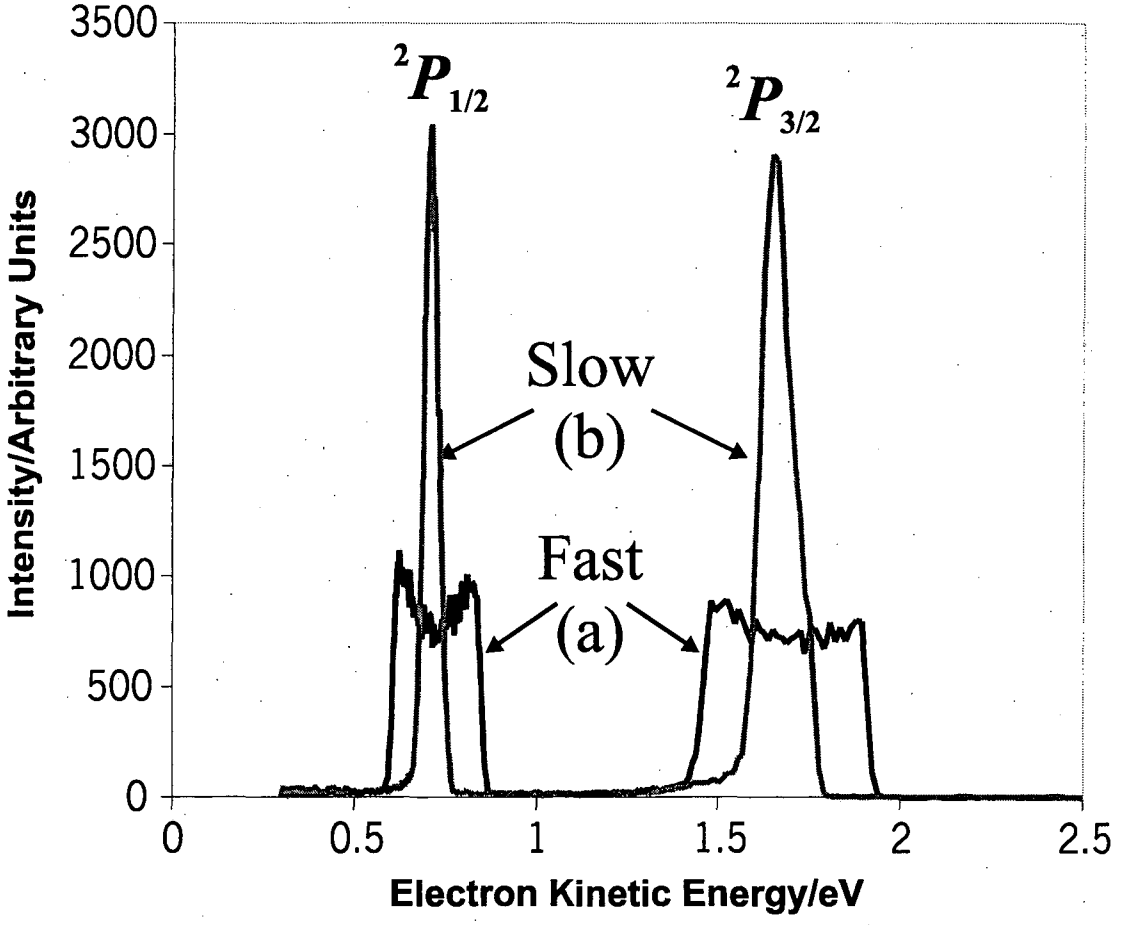


Fig. 15. Sample photoelectron spectra of I^- obtained (a) without ion deceleration, and (b) with ion deceleration.

As a demonstration of the deceleration properties of the spectrometer, Fig. 15 shows a spectrum of I^- (127 amu) measured both with and without deceleration. With a photon energy of 4.71 eV, both spectra display two peaks, at 1.65 eV ($I\ ^2P_{3/2} \leftarrow I^-\ ^1S_0$) and 0.71 eV ($I\ ^2P_{1/2} \leftarrow I^-\ ^1S_0$). The undecelerated electron kinetic energy widths are 450 and 240 meV, respectively, consistent with the known beam energy U_i = 1400 eV. The initial energy spread ΔU_i is estimated to be 100 eV. The decelerated spectrum displays energy widths of 100 and 45 meV, respectively, a roughly 5-fold decrease, indicating $U_f \approx 55$ eV and hence $\Delta U_f \approx 20$ eV.

1.9. Reflectron ion time of flight

The reflectron operates on the principle that the kinetic energy of a parent ion U_{ion} is shared among photofragments according to their masses m_{frag}, so that the velocity of each fragment v_{frag} is unchanged:

$$v_{frag} = v_{ion} \tag{22}$$

$$U_{frag} = \tfrac{1}{2} m_{frag} v_{ion}^{\ 2} = \frac{m_{frag}}{m_{ion}} (\tfrac{1}{2} m_{ion} v_{ion}^{\ 2}) = \frac{m_{frag}}{m_{ion}} U_{ion} \tag{23}$$

where v_{ion} and m_{ion} are the parent ion velocity and mass, respectively, and U_{frag} is the fragment ion kinetic energy. The uniform retarding electric field of the reflectron serves to separate ions of different kinetic energy by allowing them to penetrate to different distances r ($\leq$ 15.2 cm, the full length of the reflectron) before being reflected, exiting the reflectron with the same energy as they entered:

$$r = \frac{U_{frag}}{qE_r} \tag{24}$$

where E_r is the electric field of the reflectron. The reflectron is tilted so that the beam exits at an angle θ (9°) with respect to the incoming beam, directing the reflected ions toward an off-axis detector. The primary time-of-flight focus (at the laser interaction region) occurs at a considerable distance from the entrance to the reflectron, unlike the setup described in Lineberger and coworkers,[8] where the focus is very close to the reflectron front plate. Therefore, the time of flight T between the primary focus and arrival at the detector is modified slightly from their equation:

$$T = \frac{2m_{frag} \sec(\theta/2)}{qE_r} \left(\frac{2U_{ion}}{m_{ion}} \right)^{\frac{1}{2}} + (D_1 + D_2 \sec\theta) \left(\frac{2U_{ion}}{m_{ion}} \right)^{-\frac{1}{2}}. \tag{25}$$

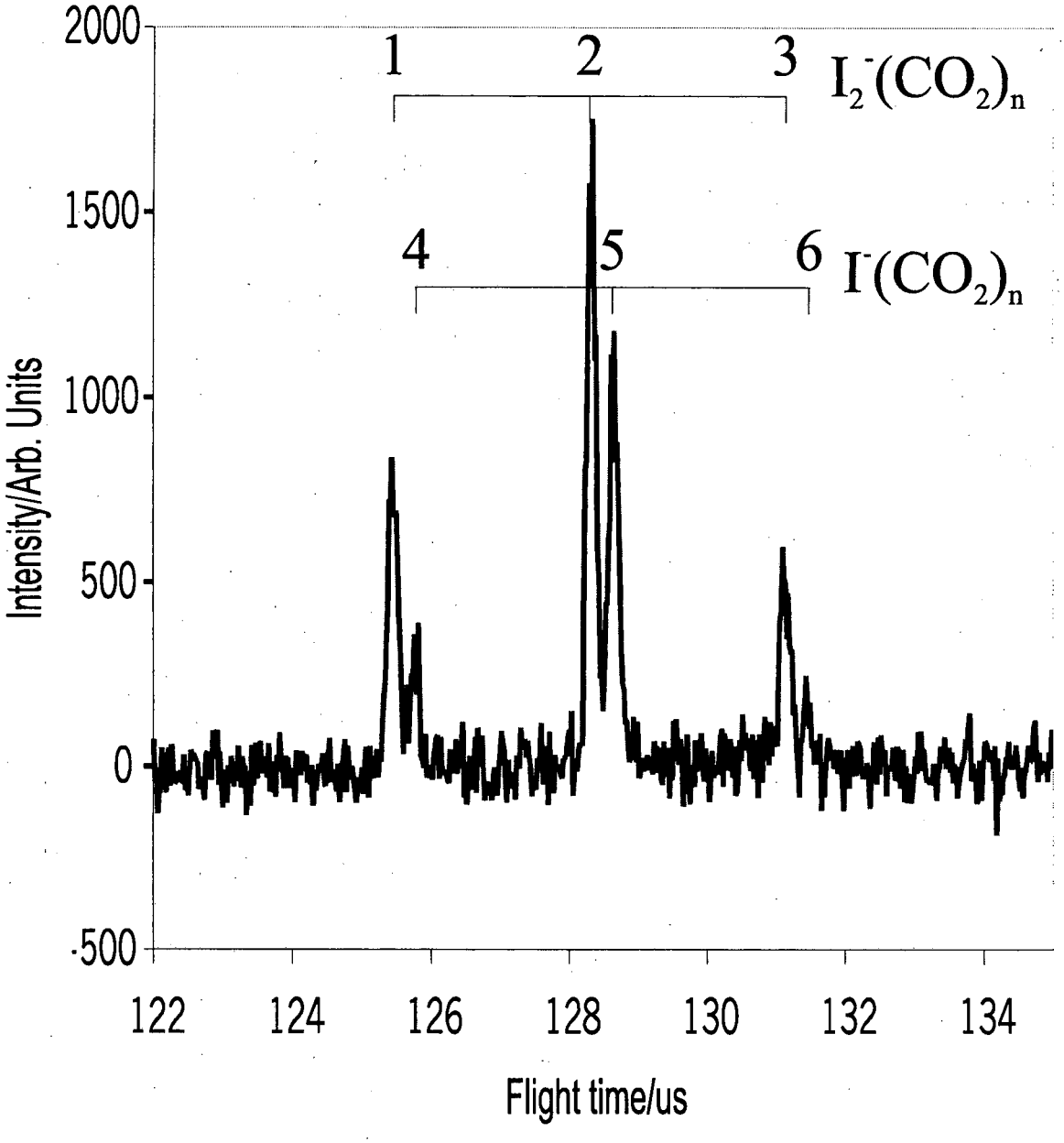


Fig. 16. Reflectron ion spectrum for the 780 nm photodissociation of $I_2^-(CO_2)_8$.

Here D_1 is the distance from the primary focus to the reflectron entrance, and D_2 is the distance from the reflectron entrance to the detector. Note that T is linearly dependent on m_{frag}. The secondary focus at the reflectron detector is obtained when $dT/dU_{ion} = 0$ (here recognizing that sec $\theta = 1.012 \approx 1$):

$$D_1 + D_2 = \frac{4U_{\text{frag}}}{qE_r} = 4r. \quad (26)$$

With D_1 = 38 cm and D_2 = 3 cm, r = 10.3 cm, about 70% of the reflectron length. Note that, for optimal focusing, E_r must be adjusted for each fragment ion.

A sample reflectron spectrum of the 780 nm photodissociation of $I_2^-(CO_2)_8$ is shown in Fig. 16. Both $I_2^-(CO_2)_n$ and $I^-(CO_2)_n$ photofragments are visible, as two sets of progressions. Because I^- (127 amu) has approximately the same mass as 3 CO_2 molecules (132 amu), the $I_2^-(CO_2)_n$ and $I^-(CO_2)_n$ peaks are quite close to one another, but they are easily resolved. The focus is optimized for $I^-(CO_2)_5$, which has a resolution $M_s \approx 130$.

2. Laser system

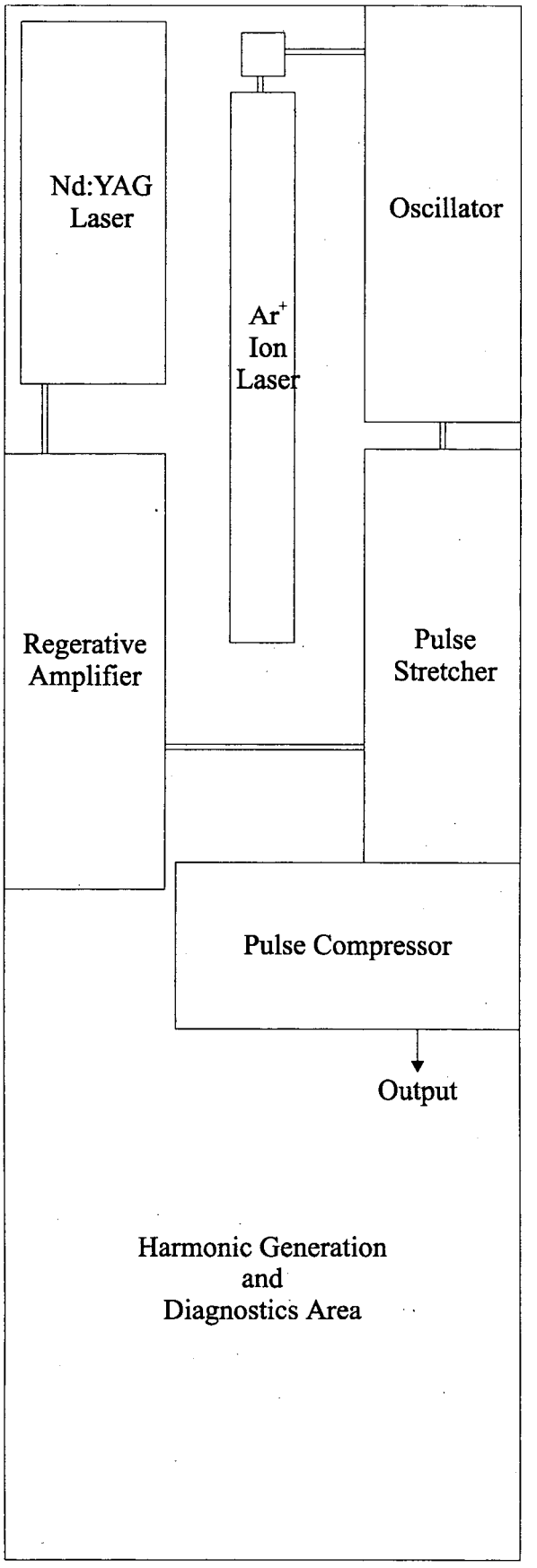
The femtosecond laser system occupies the full area of a 4 ft. × 12 ft. laser table (Newport-Klinger RS 2000). In addition to the Clark-MXR femtosecond laser, there are several other commercial and home-built components: beam pointing system, harmonic and parametric frequency generation, translation stages, autocorrelators, beam chopper, and coupling to the vacuum system. This section covers all these components in detail, as well as necessary theoretical background.

Figs. 17 and 18 show the layout of the laser table. The Clark-MXR laser system occupies approximately the full width and first (farthest from the vacuum chamber) 8 feet of length of the table. The system consists of six components, described below in more detail. After the amplified laser beam emerges from the pulse compressor of the system, it encounters beam steering optics, which include a pointing stability system called Pointmaster. The beam is then split at the first kinematically mounted, removable beamsplitter (typically 50% reflection), and the reflected light is directed toward the

fundamental frequency segment of the table. The transmitted light continues on to a second kinematically mounted, removable beamsplitter (typically 70% reflection), which directs a portion of the beam into the Quantronix TOPAS optical parametric amplifier (OPA). The light transmitted encounters additional steering optics before entering the CSK Optronics harmonic generator, which produces both second and third harmonic frequency laser light.

The fundamental frequency segment includes, first, a half-wave plate, rotating the polarization from horizontal to vertical. The beam next encounters a glass window, antireflection-coated on only one side to provide a small (4%) reflectance, mounted on a "flipper" (New Focus 9891, allowing rotation in and out of the beam with kinematic reproducibility). This reflected beam is used by two of the three autocorrelators for monitoring the temporal width of the laser pulse. Both autocorrelators share the same detector; the slow-scan autocorrelator signal is directed along the same path as the fast-scan autocorrelator using another 50% beamsplitter (not shown). The part of the beam not reflected by the glass window continues on to a flipper-mounted mirror, directing the entire beam toward the single-shot autocorrelator. When this mirror is rotated out of the way, the beam continues through a long delay line, and then into a retroflector mounted on a translation stage. After emerging from the retroreflector, it is directed through a periscope, which raises the height of the beam to the level of the vacuum chamber laser window.

Fig. 17. Laser table layout – overview (on next page).


Nd:YAG
Laser
Ar+
Ion
Laser
Oscillator
Regerative
Amplifier
Pulse
Stretcher
Pulse Compressor
Output
Harmonic Generation
and
Diagnostics Area

Scale 1:16

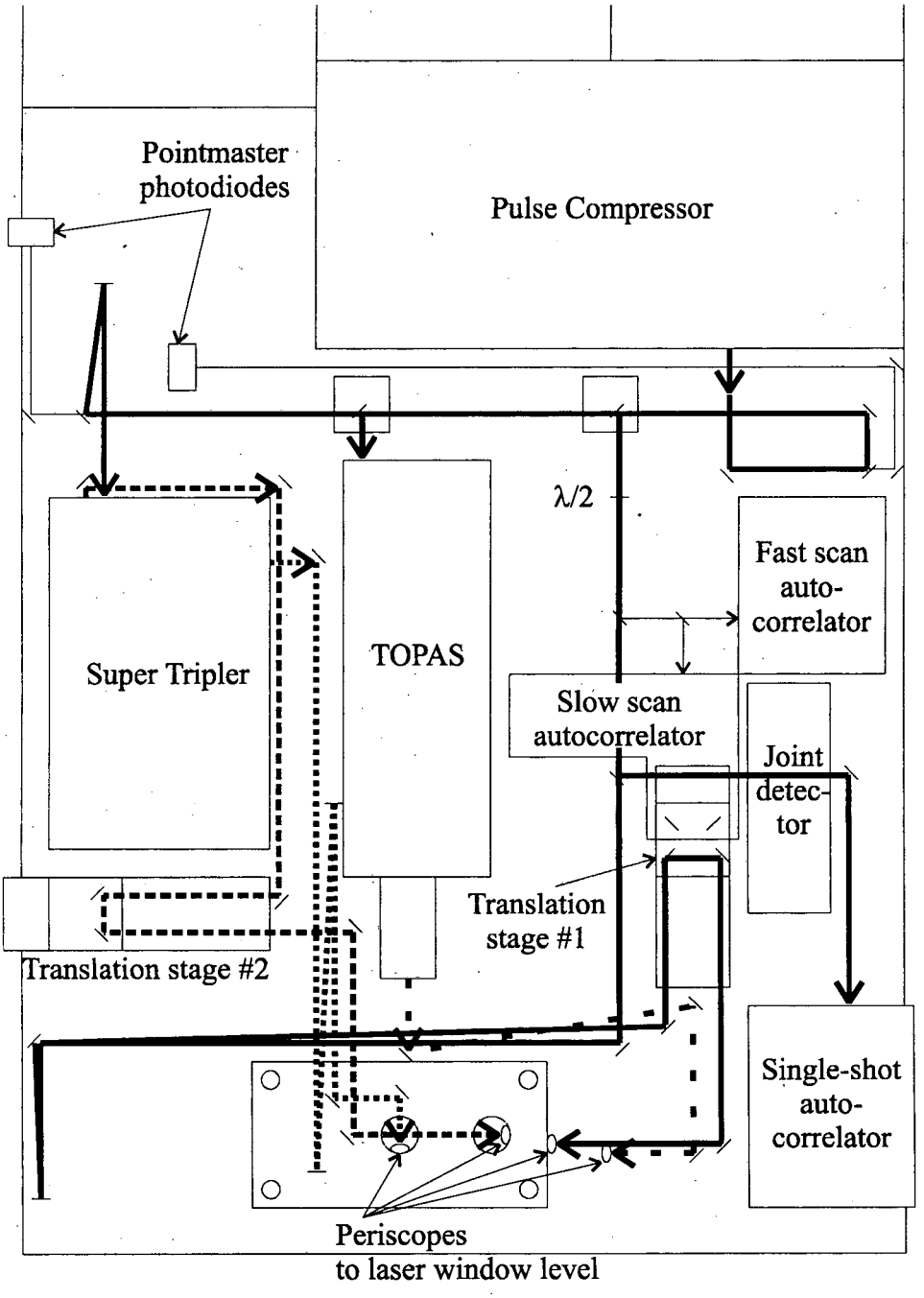


Fig. 18. Laser table layout – close-up.

The output from the TOPAS OPA encounters a pair of beam separation mirrors designed to reflect only the desired frequency. This beam is then directed into another periscope to laser window height. The harmonic generator outputs are similarly directed into periscopes, but the second harmonic frequency beam is first sent into a second, translation stage mounted retroreflector, and the third harmonic frequency beam is sent through an optical delay line prior to the periscope. The purpose of the delay lines is to synchronize the arrival time of each beam inside the vacuum chamber at some translation stage position.

Once all the beams are collinear at the laser window height, they are directed into the vacuum chamber (see Fig. 24 below). A set of beam combining mirrors allows lower frequency beams to pass through each mirror which reflects a given higher frequency beam. The beams encounter a long focal length (50 cm or 1 m), uncoated lens, just prior to passing through the laser window. On the other side of the vacuum chamber, they emerge and propagate toward the far wall of the laboratory, though usually they are blocked just beyond the exit window.

2.1. Principles of nonlinear optics

Femtosecond laser pulses are characterized by large frequency bandwidths as well as very large peak intensities, both of which tend to push their interaction with optical components into a "nonlinear" response regime. The frequency spread leads to temporal broadening and other pulse distortions, while the high intensity is responsible for efficient harmonic generation, self-phase modulation, self-focusing, and easily damaged materials. Information presented in this section is taken from both Diels and Rudolph,[9] and Shen.[10]

The electric field of the laser pulse is usually broken into a slowly-varying envelope function, and an exponential phase function, in both the frequency and time domains, related through Fourier transformation:

$$\tilde{E}(\omega) = E(\omega)\, e^{i\phi(\omega)} \tag{27}$$

$$\tilde{E}(t) = E(t)\, e^{i\phi(t)}\, e^{i\omega_0 t} = \frac{1}{2\pi}\int_{-\infty}^{\infty} \tilde{E}(\omega) e^{i\omega t} d\omega \tag{28}$$

where $\tilde{E}(\omega)$ or $\tilde{E}(t)$ is the complex electric field, $E(\omega)$ or $E(t)$ is the envelope function, and $\phi(\omega)$ or $\phi(t)$ is the phase function. In the time domain, the carrier frequency ω_0 is used to remove the high-frequency oscillations in the phase, and is somewhat arbitrarily chosen; for symmetric pulses (Gaussian or sech profile), it is usually the average frequency.

It is convenient to refer to the phase in terms of elements of a Taylor expansion:

$$\phi(\omega) = \sum_{n=0}^{\infty} \frac{1}{n!} \phi^{(n)}(\omega)\Big|_{\omega_0} (\omega - \omega_0)^n \tag{29}$$

$$\phi(t) = \sum_{n=0}^{\infty} \frac{1}{n!} \phi^{(n)}(t)\Big|_{t_0} (t - t_0)^n \tag{30}$$

where $\phi^{(n)}(\omega) = d^n\phi/d\omega^n$ and $\phi^{(n)}(t) = d^n\phi/dt^n$ are the nth derivative functions, and t_0 is, like ω_0, a reference point centered in time on the pulse. In the absence of nonlinear distortions, an optimally shaped femtosecond pulse will have contributions from the $n = 0$ and $n = 1$ terms only. $\phi^{(2)}$ is referred to as the "chirp" of the pulse, which is sometimes expressed in terms of an instantaneous frequency:

$$\omega(t) = \omega_0 + \phi^{(1)}(t) \tag{31}$$

Thus, $d\omega(t)/dt = d\phi^{(1)}(t)/dt = \phi^{(2)}(t) \neq 0$ for chirped pulses, indicating that the instantaneous frequency changes with time. "Up-chirp" refers to an increasing frequency sweep or $\phi^{(2)}(t) > 0$, while "down-chirp" refers to $\phi^{(2)}(t) < 0$. Higher-order phase distortions, e.g. $\phi^{(3)} \neq 0$ etc., are also possible.

The propagation of light through a medium is only possible because of the "sympathetic" oscillation of atoms or molecules in response to the incoming electromagnetic field. This response is expressed by the polarization **P** of the medium. In the limit of a weak radiation source at a frequency far from any resonances, **P** is linearly proportional to the vector electric field **E**:

$$\mathbf{P}(\omega) = \chi^{(1)}(\omega)\cdot\mathbf{E}(\omega) \tag{32}$$

where $\chi^{(1)}(\omega)$ is the (3×3 matrix) linear polarizability of the medium, related in the isotropic case to the index of refraction $n(\omega)$ by:

$$n^2(\omega) = 1 + 4\pi\chi^{(1)}(\omega). \tag{33}$$

For a sufficiently strong incoming field, this linear approximation breaks down, and nonlinear dependencies on the electric field must be considered:

$$\mathbf{P}(\omega) = \chi^{(1)}(\omega)\cdot\mathbf{E}(\omega) + \chi^{(2)}(\omega):\mathbf{E}(\omega)\mathbf{E}(\omega) + \chi^{(3)}(\omega):\mathbf{E}(\omega)\mathbf{E}(\omega)\mathbf{E}(\omega) + \ldots \tag{34}$$

where $\chi^{(2)}$, $\chi^{(3)}$, etc. are higher-order tensor polarizabilities of the medium. These higher-order terms are responsible for radiation at new frequencies (2ω, etc.), and will be dealt with in the section on harmonic generation. For a femtosecond laser pulse, propagation through a medium in the linear coupling limit still produces nonlinear changes in pulse characteristics, owing to the frequency dependence of $\chi^{(1)}$ or n.

Generally, the electric field is expressed in terms of z, the position inside the medium, and ω:

$$\tilde{E}(\omega, z) = \tilde{E}(\omega, 0)\, e^{-ik(\omega)z}. \tag{35}$$

Here $k(\omega)$ is the wave index:

$$k(\omega) = \frac{\omega n(\omega)}{c} \tag{36}$$

where c is the speed of light in vacuum. A number of useful quantities can be defined based on derivatives of $k(\omega)$. The group velocity, v_g:

$$\frac{1}{v_g} = \left.\frac{dk}{d\omega}\right|_{\omega_0} = \frac{n}{c} + \frac{\omega}{c}\left.\frac{dn}{d\omega}\right|_{\omega_0} \tag{37}$$

describes the average speed of the pulse inside the medium; when $dn/d\omega$ is small, $v_g \approx c/n(\omega_0)$. The group velocity dispersion or GVD, is defined as:

$$\frac{dv_g}{d\lambda} = \left.\frac{\omega^2 v_g^{\,2}}{2\pi c}\frac{d^2k}{d\omega^2}\right|_{\omega_0} = \left.\frac{\omega^2 v_g^{\,2}}{2\pi c}\left(\frac{2}{c}\frac{dn}{d\omega} + \frac{\omega}{c}\frac{d^2n}{d\omega^2}\right)\right|_{\omega_0} \tag{38}$$

where λ is wavelength. The effect of GVD on a pulse generally results in chirp. For instance, a pulse passing through a medium of length z acquires a phase

$$\phi(\omega, z) = \phi(\omega, 0) + k(\omega)z \tag{39}$$

with

$$\phi^{(2)}(\omega_0, z) = z\left.\frac{d^2k}{d\omega^2}\right|_{\omega_0} \neq 0. \tag{40}$$

For an initially unchirped, Gaussian-shaped pulse of form

$$E(t) = E_0\, e^{-(t/\tau_G)^2} \tag{41}$$

where E_0 is the maximum field intensity, and τ_G is related to the pulse duration τ_p (FWHM) by

$$\tau_p = \sqrt{2\ln 2}\,\tau_G \approx 1.177\,\tau_G\,, \tag{42}$$

we find that τ_G increases with z as

$$\tau_G(z) = \tau_G \sqrt{1 + \left(\frac{2\left|\phi^{(2)}(\omega_0, z)\right|}{\tau_G^{\,2}} \right)^2} \quad . \tag{43}$$

Thus, so long as $2|\phi^{(2)}(\omega_0, z)| << \tau_G^{\,2}$, the pulse broadening is not significant. For input pulses of $\tau_G \approx 100$ fs, $\phi^{(2)}$ is generally negligible for a few mm of material at infrared frequencies. However, it becomes significant in the ultraviolet. For instance, in fused silica at 250 nm, $\phi^{(2)} = 2800$ fs^2/cm, producing a pulse width of $1.15\tau_G$ for $z = 1$ cm.

Third-order dispersion, based on $d^3k/d\omega^3$, is also important for pulses with durations shorter than ~50 fs. Although minor for our laser system, it nevertheless plays a role in optimal pulse compression, which is discussed in section 2.9.

2.2. Clark-MXR femtosecond laser

The amplified femtosecond laser source (Clark-MXR CPA-1000) consists of six components housed in separate boxes (see Fig. 17): an Ar^+ ion laser (Coherent Innova I90-6), femtosecond oscillator (Clark-MXR NJA-5), pulse stretcher (Clark-MXR PS-1000), Nd:YAG laser (Clark-MXR ORC-1000), regenerative amplifier (Clark-MXR TRA-1000), and pulse compressor (Clark-MXR PC-1000). Both the oscillator and regenerative amplifier use a Ti:sapphire crystal as the lasing medium.

The Ar^+ ion laser (continuous wave, all lines, ~2.5 W) pumps the first Ti:sapphire rod in the oscillator, generating weak (~2.5 nJ) femtosecond pulses at ~100 MHz. The basic principle of solid-state femtosecond pulse generation is self-phase modulation, arising from changes in the index of refraction at very high intensities:

$$n(t) = n_0 + n_2 |\tilde{E}(t)|^2 \tag{44}$$

where n_0 is the linear index of refraction and n_2 is the first nonlinear term. Note that $|\tilde{E}(t)|^2$ is proportional to the intensity $I(t)$, while n_2 is proportional to the third-order polarizability $\chi^{(3)}$:

$$n_2 = \frac{3\chi^{(3)}}{8n_0}. \tag{45}$$

Most materials, including Ti:sapphire, have $\chi^{(3)} > 0$. Thus, when focused sufficiently, a pulse of moderately short light will be retarded more at the peak intensity than at the fringes, resulting in a time-dependent phase:

$$\phi(t, z) = \phi(t, 0) - \frac{k_0(\omega_0)n_2(\omega_0)}{n_0(\omega_0)} z\,|\tilde{E}(t, 0)|^2 \tag{46}$$

$$k_0(\omega) = \frac{\omega n_0(\omega)}{c}. \tag{47}$$

Rather than chirping or otherwise distorting the phase of the pulse, the effect is to broaden the frequency spectrum and thereby shorten the pulse if the group velocity dispersion is compensated for (typically, using a pair of intracavity prisms). Evidence for a wider frequency sweep can be seen by examining the instantaneous frequency:

$$\omega(t, z) = \omega_0 + \phi^{(1)}(t, z) = \omega_0 + \phi^{(1)}(t, 0) - \frac{k_0(\omega_0)n_2(\omega_0)}{n_0(\omega_0)} z \frac{d}{dt}|\tilde{E}(t, 0)|^2. \tag{48}$$

We see that the maxima in $|\omega(t, z)|$ are found at the inflection points of $|\tilde{E}(t, 0)|^2$, and there the magnitude of the frequency excursions increases with larger peak power and shorter pulse duration.

Amplification of pulses is accomplished using the technique called chirped pulse amplification, or CPA (hence the name CPA-1000). This involves stretching pulses from the oscillator out in time through up-chirping, multi-pass amplification, and subsequent

recompression using down-chirping. The reason for stretching out the pulse is twofold. First, it avoids damage to the amplifier optics, which would quickly occur if an unchirped pulse were to propagate through the cavity at full power. Second, the tremendously higher peak power of an amplified femtosecond pulse would induce unwanted nonlinear effects in the gain medium, ruining the pulse duration among other properties. By carefully choosing a pulse duration short enough to preferentially stimulate emission from the gain medium over the normally occurring ns-duration pulse, yet long enough to suppress deleterious nonlinear effects, effective amplification is achieved.

Up-chirping is accomplished by bouncing pulses off a diffraction grating several times, which introduces GVD through angular dispersion, generating pulses of roughly 100 ps duration. A Pockels cell (controlled by the Clark-MXR DT-505 high-voltage electronics) is used to select pulses at ~500 Hz for injection into the regenerative amplifier cavity, which consists of a second Ti:sapphire rod synchronously pumped by the Nd:YAG laser (500 Hz, 100 ns duration, 532 nm, 10 mJ/pulse). This "seed" pulse makes several passes inside the cavity, where it quickly builds up intensity in preference to the ns-duration pulse which would normally be created every laser shot. The amplified pulse (~1.5 mJ) is switched out of the cavity *via* the Pockels cell, and directed into the compressor. Recompression is accomplished by reversing the up-chirping process with a down-chirp, using a second diffraction grating in combination with a retroreflector. Aside from some technical challenges discussed in section 2.9, transform-limited 80 fs pulses with 1.0 mJ energy can be routinely generated at up to 1 kHz repetition rate using the Clark-MXR laser. For all experiments discussed in this dissertation, however, a repetition rate of 500 Hz was used.

2.3. Beam pointing stability

The laser system has been plagued by temperature fluctuation-induced problems since it was purchased, and this has been partially, though not completely, alleviated through the use of temperature regulation in three of the four covered boxes comprising the laser system: oscillator, pulse stretcher and pulse compressor (a unit was not available for the regenerative amplifier). These simple devices, sold by Clark-MXR, consist of a series of resistive heaters in thermal contact with the aluminum breadboard of each box, and a small control box containing a thermocouple switch, electronic keypad and readout. Although the devices are only capable of heating, the temperature of the breadboards can be maintained to ±0.1°C if set to above ambient room temperature. It has been discovered, however, that setting them too far (> 4°C) above room temperature results in jittery beam output, perhaps due to vibrations caused by drawing too much current. Still, even with this temperature regulation there is a noticeable drift to the beam position over a period of hours, which has been dealt with using a different approach, detailed below.

The Pointmaster system, also manufactured by Clark-MXR, was designed for use with the Ar^+ ion laser beam, which is known to suffer from drift. Because it was felt that the stability of the oscillator was not problematic, it made more sense to purchase Pointmaster for use with the amplified laser beam output, since small changes in beam position, changing the overlap of the pump and probe beams inside the vacuum chamber, has disastrous effects on the normalization of FPES data. In fact, no alterations to Pointmaster were requested for use with the amplified beam (the mirrors supplied for 488 nm have simply been substituted with more appropriate ones), so it would still be

possible to reconfigure Pointmaster for use with the Ar^+ ion laser, if this turns out to be a larger source of drift.

The system consists of two piezoelectric *xy* tilt stages, two beamsplitters (~99% reflectance), two quadrant photodiode detectors, a control circuit, and remote display/keypad. As the beam emerges from the pulse compressor, it is reflected from first a piezoelectrically-mounted mirror, then a beamsplitter, then a second piezoelectrically-mounted mirror, then a second beamsplitter, on its way to the harmonic generator. The weak beams transmitted through each beamsplitter are directed along moderately long (~1 m) paths before striking a quadrant photodiode. These detectors are able to detect small changes in both horizontal and vertical position through the principle of current balancing. The face of each photodiode is divided into four sections called quadrants, and the beams are initially aligned so they strike precisely in the center, the same amount of light falling on each quadrant, producing an identical photocurrent. If the beams move, the photocurrents from each quadrant will no longer balance, and the system can tell in which direction the beam has moved. Note that this approach is insensitive to fluctuations in overall laser power, provided the change is not too great (the dynamic range of the photodiodes is somewhat limited). A feedback circuit then directs the piezoelectric stages to move the beams back to the centers of the detectors, restoring the current balance. The photodiodes are calibrated before use each day (to compensate for small alignment changes made elsewhere in the system), and are able to maintain good beam pointing stability indefinitely, provided the beam does not move too much, or the power fluctuate excessively (more than ~50%).

2.4. Quantronix TOPAS optical parametric amplifier

The Quantronix TOPAS optical parametric amplifier (OPA) was purchased fairly recently, and was not used in any experiment reported in this dissertation. Its detailed operation, therefore, will be covered in another dissertation (presumably that of Martin Zanni or Alison Davis). However, a few words will be said about the purpose of the device. It enables wavelength tunability over a very wide range, from the mid-infrared (2800 nm) to the blue (450 nm), with continuous coverage. Although pulse energies are considerably lower than those encountered in the fundamental or second harmonic frequency beams, they are comparable in most wavelength regimes to the pulse energy of the third harmonic frequency, with excellent pulse width characteristics. As a result, it is a valuable addition to the femtosecond laser arsenal.

2.5. CSK Optronics harmonic generator

The generation of second and third harmonic frequency laser pulses is essential to the study of most anions using FPES, as the electron binding energy (EBE) of most anions is typically larger than what can be accessed with fundamental frequency light. For the case of I_2^- (vertical EBE = 3.21 eV) and its clusters, the third harmonic frequency is needed. Therefore, the CSK Optronics 8315A Super Tripler has been a workhorse of the experiment, and its principles and operation will be discussed in detail.

2.5.1. Principles

The interaction of light in a condensed medium to generate light at new frequencies is a sometimes strange process to understand. I have found a classical anharmonic oscillator model to be the clearest example for me, though others may find

alternative models to be simpler. Shen[10] was consulted extensively in developing this section.

Outlined here is an explanation both for second harmonic generation (SHG) as well as sum- and frequency generation (SFG and DFG, respectively), using light of two frequencies, ω_1 and ω_2, interacting with an anharmonic oscillator. This oscillator consists of an electrically charged (q) mass (m) which is displaced along the x axis when subjected to a force (F) due to the oscillating electric fields of maximum strength E_1 and E_2 for the two frequencies, respectively. The natural frequency ω_0 of the oscillator is quantitatively unimportant, so long as it is different from both ω_1 and ω_2. The anharmonicity a may be either positive or negative, and supplies an asymmetry to the oscillator as in a real molecular interaction. No damping term is assumed. The differential equation governing the motion is:

$$\frac{d^2x}{dt^2} + \omega_0{}^2 x + ax^2 = F = \frac{q}{m}\left(E_1 e^{-i\omega_1 t} + E_2 e^{-i\omega_2 t} + \text{c.c.}\right). \tag{49}$$

Here t is time, and c.c. denotes the complex conjugate. Note that $a = 0$ for materials with a center of symmetry, since $F(x) = -F(-x)$ must be satisfied. a is proportional to the second-order nonlinear polarizability $\chi^{(2)}$.

The motion of the charged mass is assumed to be responsible for the re-radiation of light, which we solve for using a perturbative treatment:

$$x = x^{(1)} + x^{(2)} + \ldots \tag{50}$$

The first-order solution results in a term for each frequency component ω_i:

$$x^{(1)}(\omega_i) = \frac{(q/m)E_i e^{-i\omega_i t}}{\omega_0{}^2 - \omega_i{}^2} + \text{c.c.} \tag{51}$$

In other words, the medium re-radiates at each driving frequency, the normal case for materials without absorption. Substituting ax^2 with $ax^{(1)2}$, the second-order solution $x^{(2)}$ is obtained:

$$x^{(2)} = x^{(2)}(\omega_1 + \omega_2) + x^{(2)}(\omega_1 - \omega_2) + x^{(2)}(2\omega_1) + x^{(2)}(2\omega_2) + x^{(2)}(0) \tag{52}$$

with

$$x^{(2)}(\omega_1 \pm \omega_2) = \frac{-2a(q/m)^2 E_1 E_2 e^{-i(\omega_1 \pm \omega_2)t}}{\left(\omega_0^2 - \omega_1^2\right)\left(\omega_0^2 - \omega_2^2\right)\left(\omega_0^2 - (\omega_1 \pm \omega_2)^2\right)} + \text{c.c.} \tag{53}$$

$$x^{(2)}(2\omega_i) = \frac{-a(q/m)^2 E_i^2 e^{-2i\omega_i t}}{\left(\omega_0^2 - \omega_i^2\right)\left(\omega_0^2 - 4\omega_i^2\right)} + \text{c.c.}$$

$$x^{(2)}(0) = \frac{-a(q/m)^2}{\omega_0^2}\left(\frac{1}{\omega_0^2 - \omega_1^2} + \frac{1}{\omega_0^2 - \omega_2^2}\right) + \text{c.c.}$$

These new frequency components represent sum-frequency generation [$x^{(2)}(\omega_1 + \omega_2)$], difference-frequency generation [$x^{(2)}(\omega_1 - \omega_2)$], second harmonic generation [$x^{(2)}(2\omega_i)$], and optical rectification [$x^{(2)}(0)$]. Note that the terms scale as the square of the electric field strength. This concludes the development of the anharmonic model.

While generation of new frequency components is possible in all directions, momentum conservation requires a particular orientation of the incoming and outgoing beams: the so-called phase matching condition. This is expressed as (for SHG, $\omega_1 = \omega_2$):

$$\mathbf{k}(\omega_1) \pm \mathbf{k}(\omega_2) = \mathbf{k}(\omega_3) \tag{54}$$

or

$$n(\omega_1)\omega_1 \pm n(\omega_2)\omega_2 = n(\omega_3)\omega_3 \tag{55}$$

where $\mathbf{k}(\omega_i)$ is the momentum of the beam with frequency ω_i, and $\omega_3 = \omega_1 \pm \omega_2$ is the new frequency component. For most materials, n increases with frequency in the visible

or near-infrared range. Therefore, for sum-frequency and second harmonic generation, where $n(\omega_3)$ is larger than both $n(\omega_1)$ and $n(\omega_2)$, the equation cannot be satisfied. However, a negative birefringent material has the property that the index of refraction of a beam polarized perpendicular to the optic axis of the crystal (the "extraordinary" wave) is smaller than the index of refraction of a beam polarized parallel to the optic axis (the "ordinary" wave). The index of refraction of the extraordinary wave n_e also depends on the angle θ of the beam relative to the optic axis:

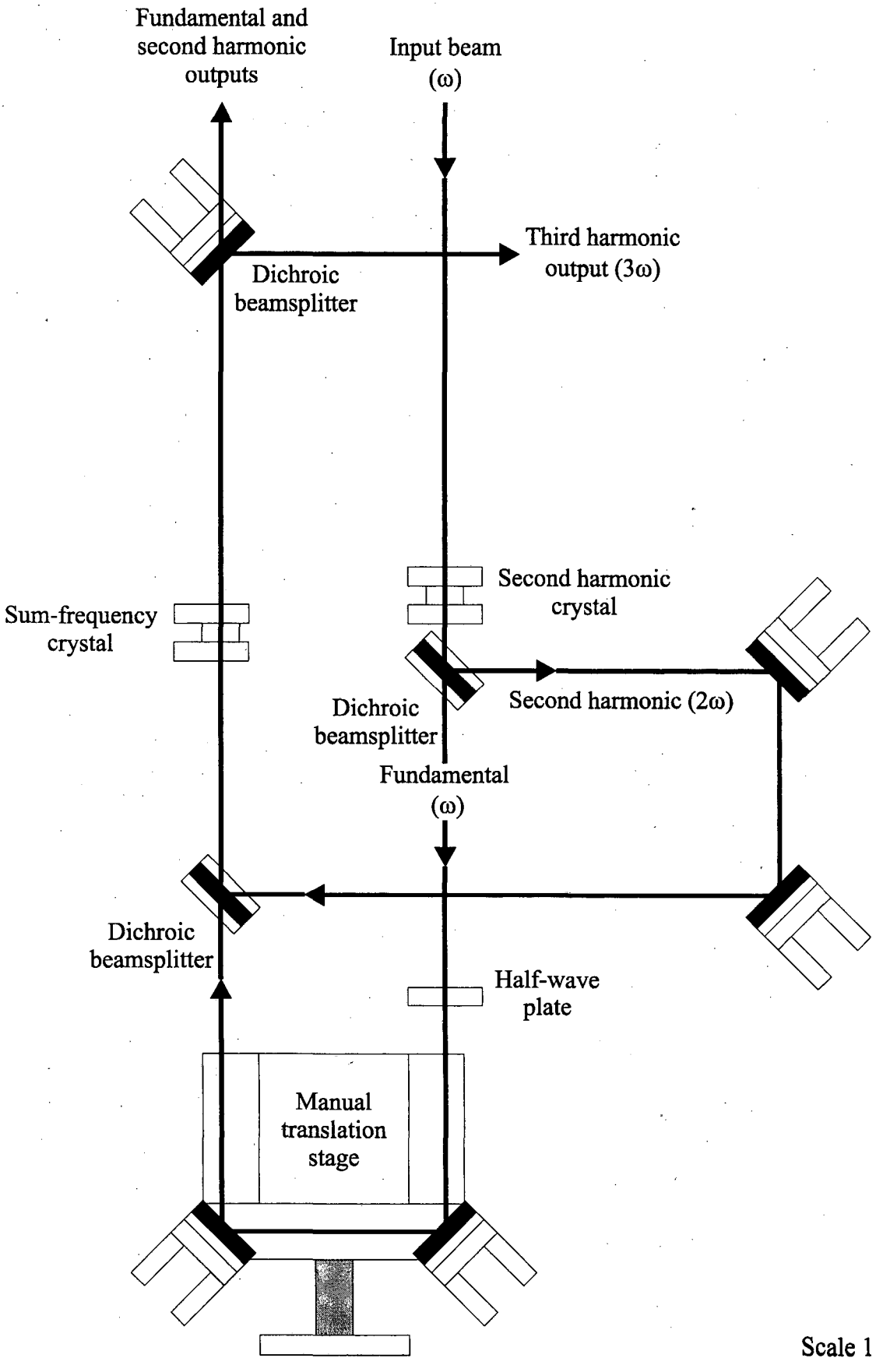
$$n_e = \frac{n_{em} n_0}{\sqrt{n_0^2 \sin^2\theta + n_{em}^2 \cos^2\theta}} \tag{56}$$

where n_{em} is the minimum in the extraordinary index of refraction, and n_0 is the ordinary index of refraction, which is independent of θ. Two polarization schemes can be used to satisfy Eq. 55, referred to as Type I and Type II phase-matching. Both require wave ω_3 to be extraordinary, but Type I requires waves ω_1 and ω_2 both to be ordinary or extraordinary, while Type II requires one of each. The CSK Super Tripler uses Type I phase-matching for generation of both second-harmonic and sum-frequency beams.

2.5.2. Operation

The layout of the harmonic generator is shown in Fig. 19. For illustration, it is assumed that 800 μJ of 780 nm fundamental light is used. The incoming fundamental frequency beam (horizontal polarization) passes first through a 1.5 mm thick lithium borate (LBO) crystal, producing a collinear second harmonic frequency beam (390 nm, ~200 μJ, vertical polarization). The frequencies are next separated at a dichroic

Fig. 19. CSK Optronics harmonic generator (on next page).


Fundamental and second harmonic outputs
Input beam (ω)
Third harmonic output (3ω)
Dichroic beamsplitter
Second harmonic crystal
Sum-frequency crystal
Second harmonic (2ω)
Dichroic beamsplitter
Fundamental (ω)
Dichroic beamsplitter
Half-wave plate
Manual translation stage

Scale 1:2

beamsplitter, which reflects the second harmonic while transmitting the fundamental. The fundamental beam continues through a half-wave plate (rotating the light to vertical polarization) to a retroreflector mounted on a manually adjustable translation stage, and finally to a second, identical dichroic beamsplitter which transmits the beam. Meanwhile, the second harmonic beam is reflected off of two fixed mirrors toward the second dichroic beamsplitter, which also reflects it. The two beams, collinear and vertically polarized, now enter a 300 μm thick β-barium borate (BBO) crystal, producing third harmonic (sum-frequency) light (260 nm, ~50 μJ, horizontal polarization). Note that, unlike the second harmonic beam, which required only a match between crystal orientation and beam direction, the third harmonic beam requires in addition the proper temporal overlap of the fundamental and second harmonic beams, accomplished by adjustment of the retroreflector. This "two-dimensional search problem" makes it a bit of a challenge to find the third harmonic beam when realigning the unit, but it is no more difficult than aligning a laser cavity, also a two-dimensional problem.

After producing the third harmonic beam, the three frequencies must be separated. Originally this was accomplished using a series of CSK Optronics-provided mirrors, but the first mirror, a simple broadband metallic mirror for reflecting all three frequencies, was easily burned by the pulse energies involved. Beam separation is now accomplished using, first, a 260 nm high reflector (CVI TLM1-260-45P-1037) to separate third harmonic from the other two frequencies, and a 390 nm high reflector (CVI TLM1-260-45P-1037) on the remaining two beams to separate off the second harmonic. The fundamental beam is not usable, the brightest spatial areas having been depleted in the conversion to the harmonic frequencies, and so is directed into a beam block. Transport

of the two harmonic beams is accomplished using additional high reflectors of the same types, in order to increase the frequency purity as much as possible. The residual amount of each harmonic beam in the other beam path at the entrance to the vacuum chamber is detectable by the eye, but of inconsequential energy.

The original unit contained a pair of lenses to reduce the beam diameter by approximately 2×, but this was found to be problematic to overlapping the second or third harmonic beam and the fundamental beam inside the vacuum chamber, due to the difference in size. Rather than squeezing maximum energy out of the harmonic generator through focusing, and then reexpanding the beams afterward, it was easier to leave all beams the same size, even if it resulted in lower energy harmonic beams. It must also be pointed out, however, that there were few lenses to spare during the "proving" phase of the experiment, so that now it would certainly be feasible to implement this scheme with the guarantee of considerable gain in pulse energies, if such were required.

2.6. Aerotech translation stages

Control of the pump-probe time delay is accomplished through use of an optical delay line involving a pair of mirrors in a retroreflecting arrangement mounted on a motorized, computer-controlled translation stage (Aerotech ATS-100-150). Although there are now two such stages on the optical table (one for delaying the fundamental beam, the other, for the second harmonic beam), there is only one controller (Aerotech U100M-A-40-F1), so that cables from each stage must be switched on the controller in order to change which stage moves.

One μm equals approximately 3.34 fs for light in air, though since the laser beam must travel toward and then away from the retroreflecting mirrors, the effective delay

time is always twice the stage movement, so in practice, 1 μm = 6.67 fs. The total range of either stage is 15 cm, or ~1 ns temporally. The resolution of the stage is reported to be 1 μm, and positioning is in fact possible in 50 nm increments, giving an ultimate time resolution of ~0.3 fs. However, there is a large inaccuracy problem of ±2-3 μm having to do with the controller, rather than the stages. The stage motor is a stepper design, meaning there is a set of four coils or "poles" arranged in a ring about the motor core. To turn the motor, alternating coils turn on and off, causing the motor to step by one pole at a time. Fractional movements can be achieved by only partially turning on or off coils, but the effect is not very linear; in other words, the motor tends to jump from one pole to the next, without a great deal of flexibility. This point aside, the inaccuracy problem stems from the fact that the power circuits supplying current to each pole are not very evenly balanced, meaning that as the motor turns, it "wobbles" irregularly, resulting in a roughly sinusoidal advance of the stage with a period of 10 μm and amplitude of 2-3 μm. Amazingly, this performance is within the specifications established by the company, which actually gives an "accuracy" of ±5 μm. In short, the wrong stage was purchased.

Happily, much of the problem has been overcome through the use of software. Because the problem is reproducible on a sub-μm scale, the stage can be precisely positioned anywhere along its wobbly path. By calibrating the apparent position to the absolute position (though the use of a fitted autocorrelation trace), the variation can be reduced to a noise level. The only problem with this method is that the offset, or "phase," of the sinusoidal cycle must be determined whenever the stage is translated by a significant amount, i.e. a few mm, because the period of the cycle is not *exactly* 10 μm. In many cases, however, precise positioning is only required very close to the zero of

time, or "t_0," location, with inaccuracy of a few fs tolerated tens of ps away from this location. Fig. 20 illustrates the effectiveness of this correction with a pair of autocorrelation traces taken with and without wobble correction.

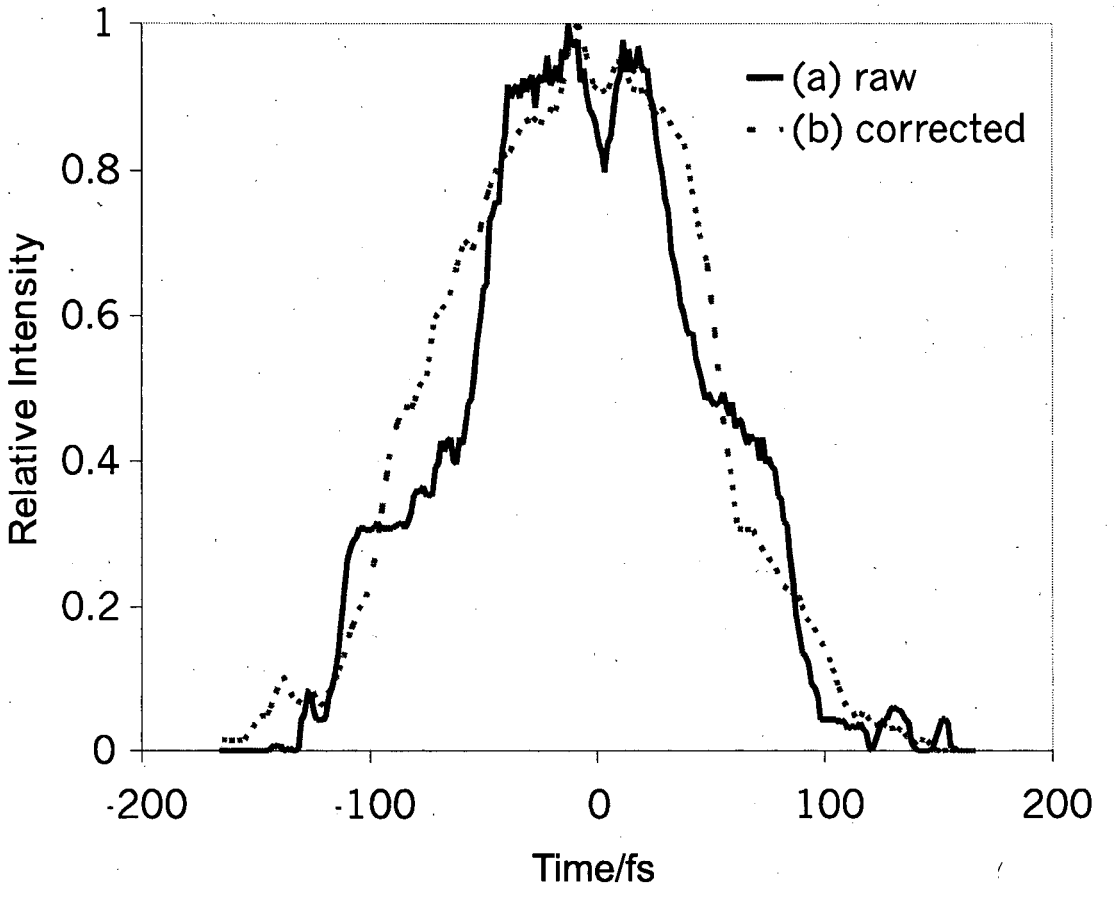


Fig. 20. Autocorrelation traces illustrating translation stage sinusoidal problem: (a) without wobble correction; (b) with wobble correction.

2.7. New Focus optical chopper

To aid the normalization of FPES spectra taken at different time delays under changing ion beam and laser power conditions, an optical chopper (New Focus 3501) was purchased to perform shot-to-shot background subtraction (see section 3.2.1). This approach entails blocking the pump laser by the blades of a rapidly spinning chopper wheel every other laser pulse, or 250 Hz, and subtracting probe-only photoelectron

spectra from pump-and-probe spectra on a per pulse basis. Synchronization of the wheel with the laser is critical, which was the most difficult feature to locate in a chopper; most models force the experiment to be triggered from the chopper itself, not an option in our experiment. The New Focus chopper may be triggered externally, and includes circuitry to run at a fractional or multiple harmonic of the trigger frequency (1/2 was required for our application). Also, the timing offset or "phase" with respect to the trigger is fully adjustable, avoiding the problem of chopping only part of a beam's spatial profile.

2.8. Autocorrelation and cross-correlation

An autocorrelation is the temporal overlap of a laser pulse with itself. Mathematically speaking, this is expressed as:

$$A(t) = \int_{-\infty}^{\infty} dt' I(t') I(t + t') \tag{57}$$

where $I(t)$ is the time-dependent intensity profile of a laser pulse, and $A(t)$ is its autocorrelation function. Although it is not possible to directly measure the intensity profile, a significant amount of information about it is contained in the autocorrelation, as can be confirmed through deconvolution. Generally, however, the most useful piece of information is simply the pulse width (full width at half maximum, or FWHM), which can be derived from the FWHM of the autocorrelation, and an assumption about the shape of $I(t)$. If it is Gaussian, then the FWHM of $A(t)$ is equal to $\sqrt{2}$ (~1.414) times the FWHM of $A(t)$. On the other hand, if the shape of $I(t)$ is sech^2, then the ratio is ~1.543. It is generally assumed[11,12] that a well-formed femtosecond pulse has a sech^2 intensity profile, though in practice, it makes little difference which is used, other than the fact that a sech^2-shaped pulse happens to have a shorter FWHM for a given $A(t)$ FWHM.

Cross correlation is analogous to autocorrelation, except that two frequencies of light are combined to generate a signal dependent on the pulse width of both pulses. Mathematically, it is very similar to Eq. 57:

$$C(t) = \int_{-\infty}^{\infty} dt' I_A(t') I_B(t+t') \tag{58}$$

where $I_A(t)$ and $I_B(t)$ are the time-dependent intensity profiles of each laser pulse, and $C(t)$ is its cross correlation. Since in principle either pulse could be characterized separately by autocorrelation, the cross correlation can be used to find the width of the other pulse though deconvolution. For gaussian pulses, the relationship is straightforward:

$$\tau_C^{\,2} = \tau_A^{\,2} + \tau_B^{\,2} \tag{59}$$

where τ_A, τ_B and τ_C are the FWHMs of the two beams and the cross correlation, respectively. For sech^2 pulses, the relationship is not analytic, but a lookup table has been employed to calculate τ_B from knowledge of τ_A and τ_C.

Measurement of the autocorrelation or cross-correlation signal is obtained through a nonlinear optical process, usually SHG for autocorrelation, and SFG or DFG for cross-correlation, inside a nonlinear crystal. See section 2.5.1 for more information on these techniques.

2.8.1. Slow-scan autocorrelator

A total of three autocorrelators have been built for the laser system. The "fast-scan" autocorrelator is used for quick, qualitative measurements only, as no data can be recorded with it. The "slow-scan" autocorrelator uses one of the motorized translation stages to vary the path length, so that a computer can direct the stage through an autocorrelation, recording the digitized photodiode signal at each step. The "single-shot"

autocorrelator is also used for qualitative measurement, but it was added in 1998 by Alison Davis, and will not be discussed here. Note that each autocorrelator is of a noncollinear type, meaning the two beams are crossed at a small angle inside the nonlinear crystal. This arrangement guarantees that the autocorrelation signal will be background free, but it also prevents an interferometric autocorrelation, containing valuable additional information, such as temporal chirp, from being obtained if a collinear arrangement were used.

The table layout for the slow-scan autocorrelator is shown in Fig. 21. After the fundamental beam is directed into the autocorrelator with a pair of beamsplitters (2% and 50%, respectively), the beam encounters a 50% beamsplitter. The reflected beam is directed toward a fixed leg retroreflector permanently mounted on the laser table, while the transmitted beam propagates toward a retroreflector mounted on the same translation stage as used for the fundamental frequency delay line (on the opposite side of the carriage). Beams are recombined with the same beamsplitter, then directed into the detector with a second 50% beamsplitter, which allows the fast-scan autocorrelator to use the detector without having to move mirrors (one pair of autocorrelator beams are blocked, of course). The detector consists of a 4" f.l. lens, a 5×5×0.5 mm BBO crystal (CSK Optronics) cut for second harmonic generation, placed just before the focal point to prevent burning, a blue bandpass filter (Newport BG40), and a photodiode (Thorlabs FDS100).

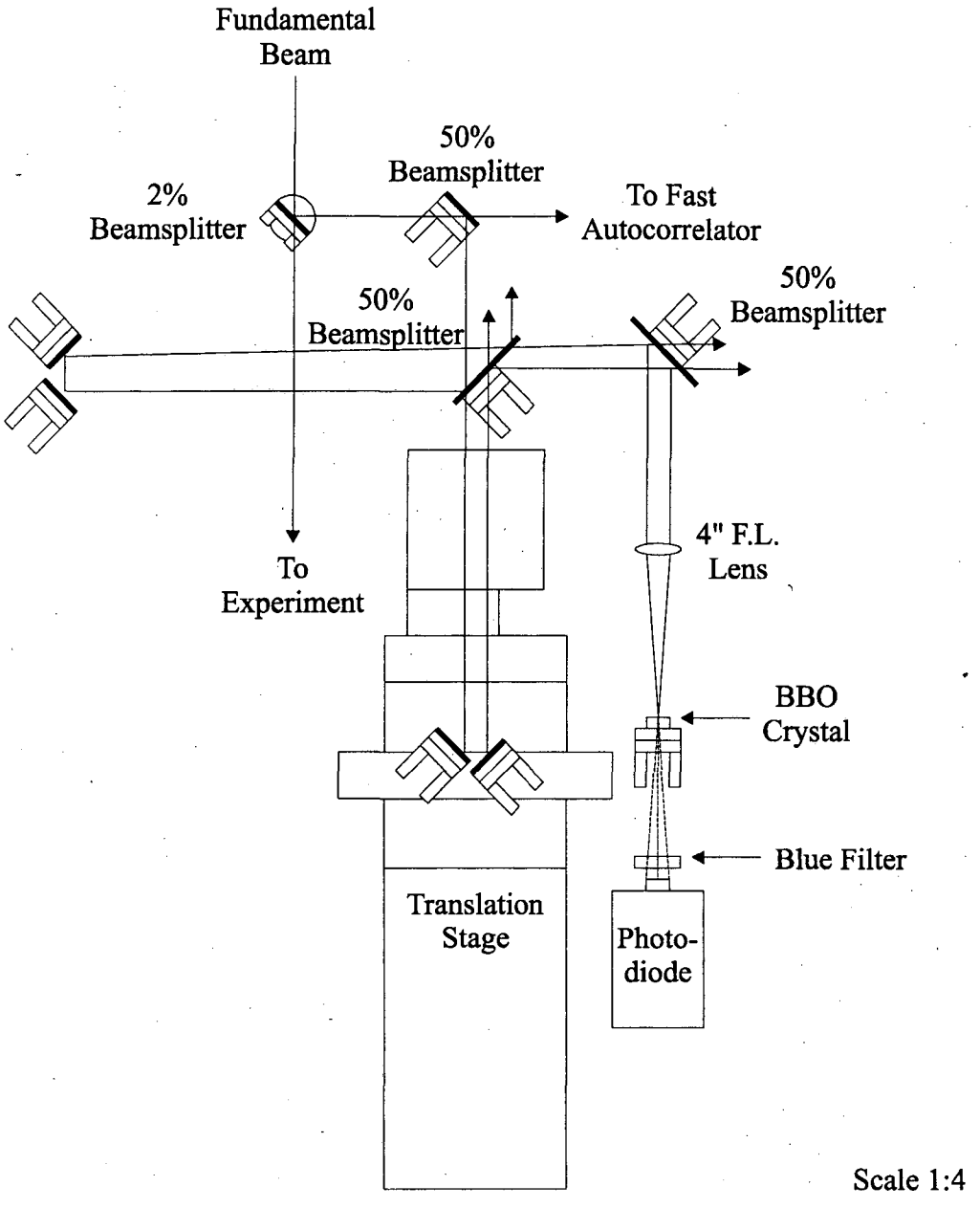


Fig. 21. Slow scan autocorrelator.

The highlight of this autocorrelator is its ability to record autocorrelation traces by computer, digitizing the photodiode signal while scanning the translation stage. An overview of the translation stage appears elsewhere in this chapter, and the digitizing electronics are described in section 3.3.

2.8.2. Fast scan autocorrelator

The layout of the fast scan autocorrelator is shown in Fig. 22. A 50% beamsplitter reflects half the beam along the variable path, which consists of a pair of mirrors mounted on a small, 90° aluminum bracket (1/16" thick) glued to a speaker cone (Radio Shack 4" woofer, part # 40-1022B), while the transmitted half travels along the fixed path consisting of a pair of mirrors mounted on a small, manual translation stage (Newport-

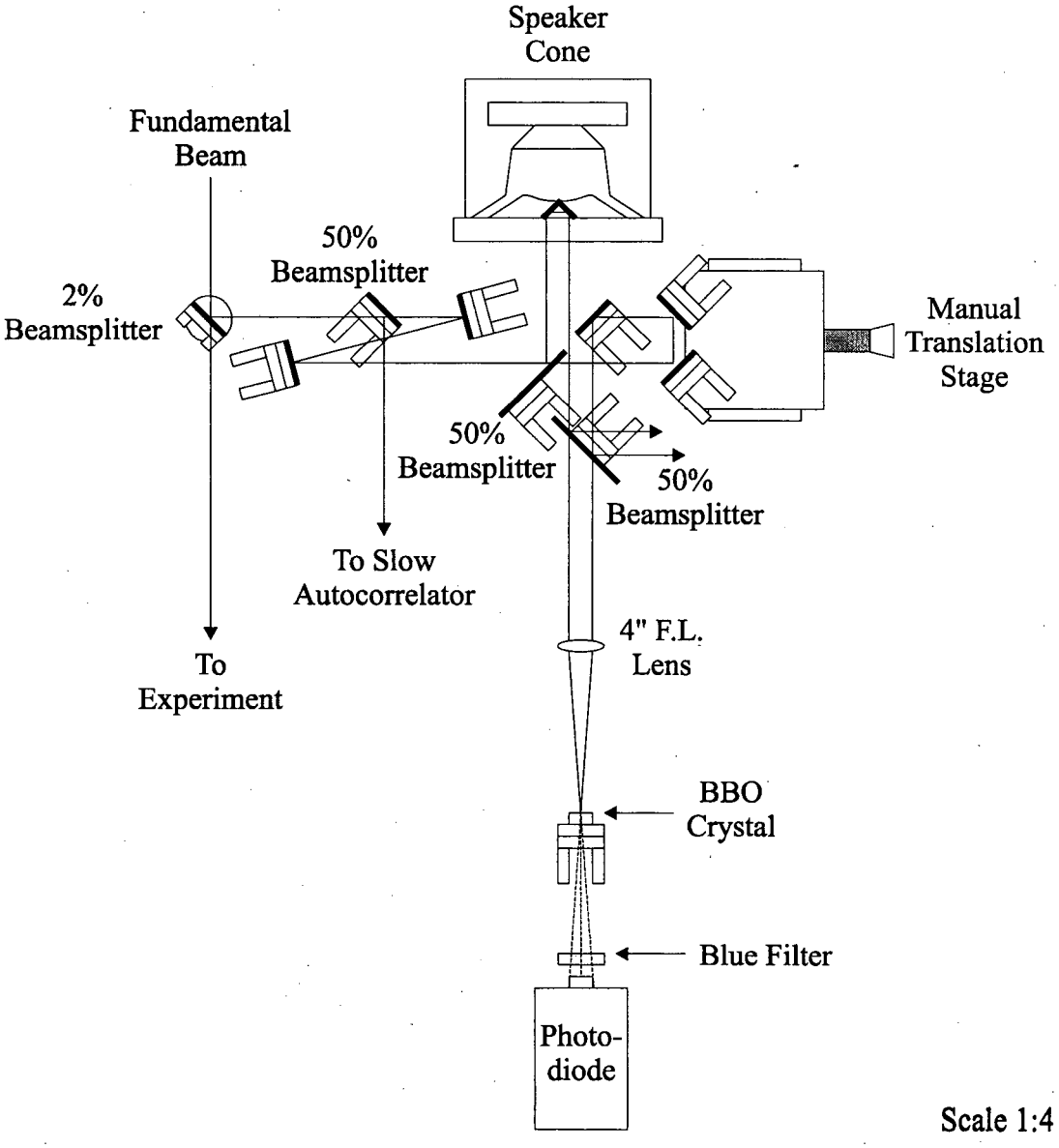
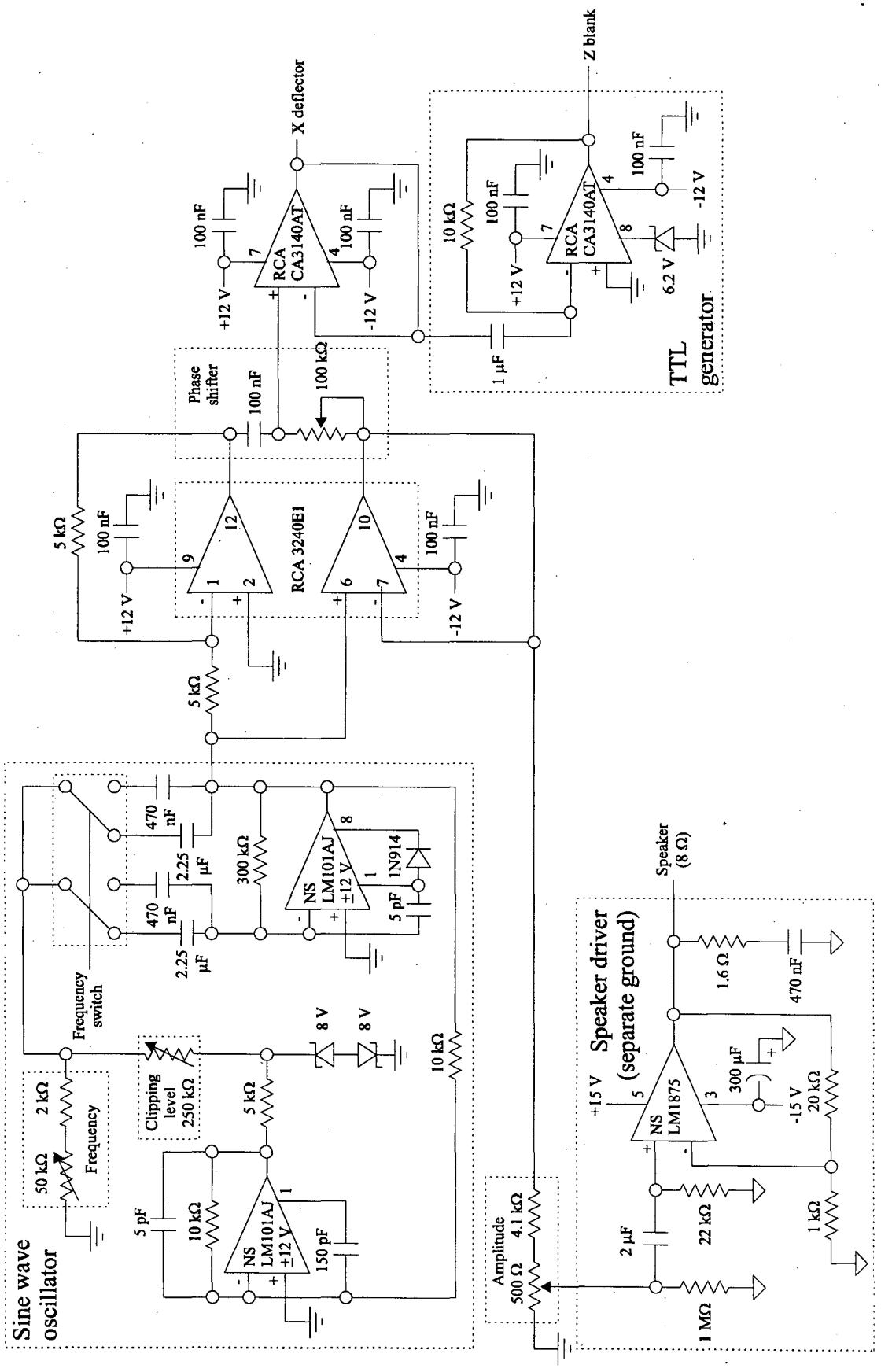


Fig. 22. Fast scan autocorrelator.

Klinger UMR5.16). Rather than recombining the beams with a beamsplitter, the variable path beam passes directly into the detector, while the fixed path beam is directed into the detector via a mirror. Both beams pass through the 50% beamsplitter used by the slow-scan autocorrelator.

The heart of the autocorrelator is the speaker-mounted retroreflector. The speaker is driven by a home-built amplifier at ~20 Hz, which scans the laser pulse from the variable path through the fixed path pulse. To obtain a real-time autocorrelation trace on an oscilloscope, the XY mode is employed, using a scaled output from the amplifier as the X input, and the photodiode signal as the Y input. Note that a relatively large impedance (1 MΩ) must be used for the photodiode signal in order to generate a slow (~100 μs) decay; otherwise, the signal will be invisible on the time scale of the laser repetition period (2 ms). Since there is a phase lag between the signal sent to the speaker and the speaker position, the scaled amplifier output has a variable phase adjustment. Also, the photodiode signal is blocked on the oscilloscope during half the cycle, in order to prevent two superimposed images from appearing, which are difficult to align; a "Z blank" output from the amplifier is used for this purpose.

Fig. 23. Fast scan autocorrelator oscillator circuit (on next page).


Sine wave oscillator
Frequency
50 kΩ
2 kΩ
Clipping level 250 kΩ
Frequency switch
470 nF
2.25 µF
300 kΩ
NS LM101AJ ±12 V
5 pF
1N914
8 V
10 kΩ
5 kΩ
150 pF
Amplitude 500 Ω
4.1 kΩ
Speaker driver (separate ground)
Speaker (8 Ω)
NS LM1875
+15 V
-15 V
300 µF
1.6 Ω
470 nF
22 kΩ
20 kΩ
2 µF
1 MΩ
1 kΩ
Phase shifter
100 nF
100 kΩ
RCA 3240E1
+12 V
-12 V
RCA CA3140AT
X deflector
Z blank
TTL generator
6.2 V
1 µF

The design of the circuit was copied from Lucas Hunziker, a graduate student with Prof. Yongqin Chen at Berkeley; it is reproduced in Fig. 23. It consists of, principally, a sine-wave oscillator, two signal-modifying circuits generating the X and Z blank outputs, and a power amplifier for the speaker. The frequency of the oscillator can be adjusted with a potentiometer and a properly chosen capacitor; with a toggle switch to choose between two sets of capacitors, the frequency range is currently 4-85 Hz. Part of the oscillator circuit is a feedback loop called "clipping," containing a potentiometer whose value must be adjusted for different frequencies, and often again once the circuit has warmed up. Setting the value too high allows the circuit to swing to the minimum and maximum voltages of the operational amplifier (op-amp), causing flattening of the sine wave; setting it too low attenuates the oscillator completely. The output from the oscillator is split; one half passes through a phase shifter, adjusted with a potentiometer, and on to two simple op-amp circuits, one of which provides the reference X signal for the oscilloscope, the other of which generates the Z blank (TTL) signal. The other half of the oscillator output is connected to a power amplifier circuit, with adjustable amplitude, and from there, to the speaker (8 Ω impedance).

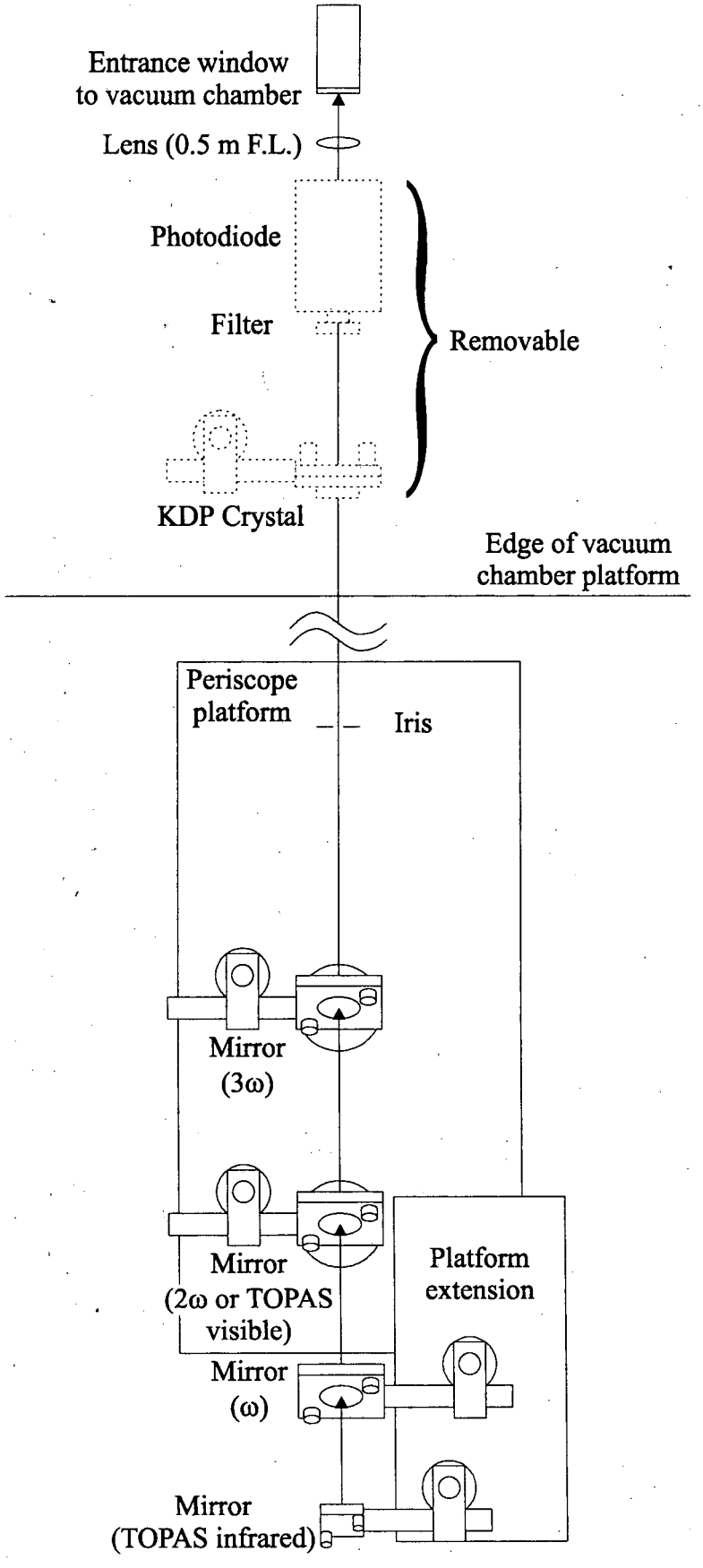
Originally, the circuit was plagued by a terrible noise problem, which seemed to stem from the use of a single power supply for all components. This problem was solved by using two power supplies, one for the power amplifier (International Power HAA15-0.8-A, ±15 V at 0.8 A), and a second for the rest of the circuit (International Power IHTAA-16W, ±12 V at 0.4 A). The addition of large capacitors on the power supply outputs also helped reduce the noise. The amplifier segment of the circuit is also separately grounded, to minimize potential ground loops.

2.8.3. Cross-correlation

The layout of a typical cross-correlation setup is shown in Fig. 24. It is a collinear arrangement, the path length of one beam being varied by a translation stage elsewhere on the laser table. Beams are combined using a mirror which allows the lower frequency beam to pass through. No focusing is required, as the full power of each beam is generally used, providing more than enough cross correlation signal intensity.

The cross correlation signal is generated using a nonlinear crystal to generate sum- or difference-frequency radiation. We have successfully measured cross correlation signals for all combinations of fundamental (ω), second harmonic frequency (2ω) and third harmonic frequency (3ω) light, using a 7×7×0.3 mm KDP crystal (CSK Optronics), cut for $\omega + 2\omega$ sum frequency generation. For the other two frequency combinations, difference frequency generation was used at the same incidence angle to the crystal; however, the polarizations of the beams had to be correctly oriented. Separation of frequencies is essential to detection in this collinear design. For ω and 2ω light, a simple filter (red or blue bandpass) before the photodiode was used to block the other frequencies. For 3ω light, a Pellin-Broca prism was employed to separate the beams spatially, and a fluorescent card placed at right angles to the photodiode was used to image the ultraviolet 3ω light, since the photodiode was insensitive to this wavelength. Recording of cross correlation spectra was accomplished in the same manner as for autocorrelation spectra.

Fig. 24. Cross-correlation optical layout (on next page).


Entrance window
to vacuum chamber
Lens (0.5 m F.L.)
Photodiode
Filter
Removable
KDP Crystal
Edge of vacuum
chamber platform
Periscope
platform
Iris
Mirror
(3ω)
Mirror
(2ω or TOPAS
visible)
Platform
extension
Mirror
(ω)
Mirror
(TOPAS infrared)

Scale 1:4

In our experiment, cross correlation is also very useful for finding the zero of time (t_0) of two beams prior to entering the vacuum chamber. Although the t_0 changes slightly each time the beams pass through additional material, such as a lens or laser window, the offset is only a few ps for 260 nm light. Therefore, cross correlation is the primary means of temporal beam alignment when preparing for an experiment, and can often be used in lieu of an actual determination of t_0 in vacuum (see section 3.2.2), once the time offset has been measured.

2.9. Laser pulse optimization

Over the course of learning how to operate the Clark-MXR system, several "tricks" have been discovered which are useful in the optimization of laser pulses. Some of the most important are detailed here.

The oscillator is remarkably stable, though over a period of weeks the average power in the cavity declines to the point where spontaneous loss of mode-locking occurs. This may be due to changes in the direction of the Ar^+ ion pump beam, but as this is not easily adjusted without a complete realignment of the oscillator, the cavity end mirrors are typically adjusted instead. The most significant improvement in power is often made with vertical adjustments, which are more sensitive. Dusting the cavity mirrors may also make large improvements in power. The continuous-wave (cw) cavity power may be optimized before achieving mode-locked operation, since the cavity alignment of the two modes is very similar. With an Ar^+ laser aperature size of 4 and pump power of 2.5 W, good alignment should approach 200 mW or more. If this benchmark cannot be reached, a general alignment of the cavity may be necessary.

The Ar^+ laser tube has a lifetime of approximately 2000 hours, or 1.5 years depending on usage, after which a common sign of impending failure is mode degradation, caused by physical descent of the electrodes within the laser tube which partially blocks the beam. This seriously affects the efficiency of the oscillator, which requires a very good quality pump mode (TEM_{00}) to achieve proper self-focusing. The mode may be easily examined by reflecting the beam, with aperature fully open, off of a concave mirror (the "mode tool" of the oscillator may be used) onto a surface several meters away. The presence of dark areas or pronounced rings usually indicates mode degradation.

The angular misalignment of the Pockels cell in the regenerative amplifier causes an incomplete polarization change to the amplified pulse train before and/or after the selected pulse has exited the cavity. This results in a series of "pre-pulses" and/or "post-pulses" in addition to the main pulse. For gross misalignments, such additional pulses can be seen in a photodiode signal from the compressor. They also appear in the autocorrelation spectrum as "wings" (temporally wide, low-intensity shoulders on an otherwise nearly $sech^2$-shaped pulse), because they are not optimally compressed, spending a different number of round trips in the amplifier cavity and therefore having a different amount of group velocity dispersion relative to the main pulse. However, other optical elements can also produce wings (see below). An unambiguous method of detecting the presence of additional pulses is by using the photoelectron spectrometer. In performing a pump-probe experiment on I_2^- (using 780 nm for the pump pulse and 260 nm for the probe pulse), it was discovered that a significant (as much as 20%) amount of dissociated I^- was being produced even at "negative" time delays, i.e. when the probe

pulse arrived before the pump pulse. Although both pre-pulses, which introduce a premature pump, and post-pulses, which supply a late probe, could be responsible for the effect seen in the photoelectron spectrum, it is most likely due to additional pre-pulses. This is because the intensity of the additional pulses should be greatly reduced for the second- and third-harmonic beams, which scale inversely with pulse duration. Since the Γ signal is very strong, adjustments made to the Pockels cell are easily detected, and the effect can be minimized, although not always completely eliminated.

Although compression of amplified pulses to ~80 fs is usually achievable without much effort, the presence of wings in the fundamental beam, as seen in an autocorrelation spectrum, is a more difficult problem to remedy. Unlike the wings caused by additional pulses, these are due to contributions from parts of the frequency spectrum which have acquired a significant third-order dispersion $\phi^{(3)}(\omega)$, and possibly higher dispersions, in the amplification process. These temporal distortions cannot be compensated for by optimizing the compressor distance, which only alters the group velocity dispersion $\phi^{(2)}(\omega)$. However, changing the compressor grating angle in the plane of the table introduces third-order dispersion which can reduce the high-order phase offset. Another approach is to introduce third-order dispersion before regenerative amplification, using the "third order knob" in the stretcher. Here, instead of changing the grating horizontal angle, the horizontal folding mirror angle is adjusted so that the reflected spectrum striking the grating is offset horizontally. High-order dispersion introduced in the amplification process will then be partially cancelled by the initial third-order dispersion. In practice, iterative adjustment of both optical elements can significantly reduce wings in a spectrum.

3. Data acquisition

3.1. Mass spectra

Mass spectra are acquired using the Tektronix TDS744A digitizing oscilloscope. This device is remotely controlled by computer using the National Instruments GPIB interface. It is capable of very high resolution (< 1 ns) data acquisition, though in practice, only 20-100 ns time intervals are used. Spectra are typically averaged for 1000-4000 scans before downloading the data. Although limited to ~80 Hz repetition rate (it is even lower when several processes are active), high-quality spectra may be obtained in only a few seconds. Up to 500,000 consecutive time intervals may be stored in the oscilloscope, though currently the PC data acquisition software is only capable of reading 1024 intervals at a time; therefore, to record larger record sizes, spectra must be obtained in 1024-interval segments. The sample mass spectrum shown in Fig. 14(a) consists of three such segments recorded consecutively, using a 20 ns time interval.

3.2. Photoelectron spectra

Electron time-of-flight spectra are acquired using the Stanford Research Systems SR430 multichannel scalar (MCS), which is a combination of discriminator and event counter. It has a > 1 kHz maximum repetition rate, 5 ns minimum time resolution, storage capacity of 32,768 consecutive time intervals, and maximum run time of 65,536 trigger events before downloading is necessary. Generally, spectra are acquired at 500 Hz using 5 ns resolution, –10 mV discrimination threshold, 1024 time intervals (total acquisition time of 5.12 μs), and data are acquired for 10,000 laser shots (20 s) before downloading.

Communication with the lab computer occurs through the National Instruments GPIB interface.

The MCS must operate in a low-signal limit, that is, in a regime where the likelihood of two electrons arriving in the same 5 ns time interval is practically zero. This requirement is in fact for a 10 ns interval, because the MCS cannot register two consecutive 5 ns events (for larger interval sizes, this limitation is absent; presumably, some of the internal circuitry has a 10 ns cycle time). However, most users would be more than happy to reduce their signal level if they discovered it to be close to 1 e^-/10 ns; with an overall 500 Hz repetition rate, this represents a very fast data acquisition rate indeed, as a high-quality spectrum requires only ~1000 e^- per time interval. More typical operating conditions are nowhere near this limit, except perhaps occasionally when photodetaching I^-.

Electron spectra may also be acquired using the Tektronix TDS744A digitizing oscilloscope. Although the resolution of this device is considerably higher (< 1 ns) than the MCS, its major limitation is the low ~80 Hz repetition rate. However, its primary application, the collection of 1-photon photoelectron spectra in shot-to-shot background subtraction mode (see section 3.2.1), does not need a large signal. To collect electron spectra, the oscilloscope is "tricked" into operating in a kind of discrimination mode whereby the baseline is shifted far offscreen, just outside the range of the digitizer, so that the spectrum appears flat except for real signals which protrude into the valid range. Note that, unlike the MCS, signals are not simply counted (as a 1 or 0), but have a variable height since their voltages are digitized with some resolution. This approach adds to the noise of a spectrum, and also presents a greater challenge when normalizing spectra

collected by the two techniques, since intensities must be scaled. Another difference of the oscilloscope technique is that consecutive spectra are averaged (with 16 bits of resolution), rather than summed, so that data must be downloaded well before the signal size of a single electron is reduced to one bit. Generally only ~1200 spectra are acquired in the time it takes the MCS to acquire 10,000, so there is plenty of resolution to spare.

3.2.1. Background subtraction

Subtraction of one-photon "background" signal is essential, where the probe laser (and, sometimes, pump laser) can detach an electron from almost every anion studied by FPES. Also, because the ion and/or laser sources do not always operate with steady intensity, concurrent measurement of background and two-photon signals is necessary to normalize two-photon spectra taken at different pump-probe time delays. In rare cases, such as ATD measurements on I^- (see section 3.2.2) the one- and two-photon signals are both present in the same spectrum at well-separated energies, and separate background subtraction is not necessary.

Two background subtraction methods have been developed, each of which has different strengths and weaknesses. The first method, "alternating scan" background subtraction, consists of switching the time delay (by moving the translation stage) between a time of interest and a fixed, negative time (that is, probe pulse before pump pulse) at typically 20 s intervals. By choosing a negative time delay much larger than the pump-probe overlap, no signal arising from anions excited by the pump laser will be present, and a one-photon, background photoelectron spectrum will be recorded. The second method, "shot-to-shot" background subtraction, is much more rapid, and uses an optical chopper (see section 2.7) to block the pump laser by the blades of a rapidly

spinning wheel every other laser pulse, subtracting probe-only photoelectron spectra from pump-and-probe spectra at 250 Hz. The MCS normally used for recording of photoelectron spectra supports this capability. However, since the integrated intensity of the time-dependent signals are not necessarily the same, separate recording of the probe-only spectra is also required to normalize spectra. An effective, if perhaps inelegant, system has been implemented using the Tektronix digitizing oscilloscope to record probe-only spectra at ~80 Hz. Although considerably less signal is recorded this way, only the integrated intensity is required to normalize spectra.

Alternating scan background subtraction is only useful when the ion and laser power intensities are not fluctuating very much. Because several time delays are generally acquired in the same set of scans, choosing a different time delay after each background spectrum, and cycling through the full set many times, the number of background scans can be much greater than the number of individual two-photon spectra. This situation allows for an advantage over shot-to-shot background subtraction: the number of laser shots used to acquire each background scan can be reduced, so that more time is spent acquiring two-photon spectra, yet a high-signal background spectrum is also obtained, averaged for all the time delays measured in the set. With shot-to-shot background subtraction, 50% of the spectra must be background, unless the chopper wheel spacing is altered. However, the main advantage of shot-to-shot background subtraction is its ability to ride out large fluctuations in ion and laser intensity. While it is always advisable to correct such problems before acquiring data, there is almost inevitably a slow drift to the ion intensity over a period of minutes, which is not easily

compensated for by the alternating scan method, unless a large number of scans (> 20) are acquired at each time delay.

3.2.2. Above threshold detachment

As mentioned in the section on cross-correlation, when using two laser frequencies in an experiment, t_0 changes each time the beams pass through material, such as a lens or laser window. The pulse width of each beam also increases, much more so for shorter wavelengths. For both these reasons, it is essential to be able to determine t_0 inside the vacuum chamber. The nonresonant, two-photon above-threshold detachment (ATD) of a halide[13] such as I^- has been used successfully for this purpose. The technique is general, but most commonly consists of using the fundamental (780 nm) and third-harmonic (260 nm) beams. When beams are temporally separate, only the 260 nm light detaches electrons, but when the beams are temporally overlapped, new photoelectron features appear at higher kinetic energy, corresponding to two-photon detachment. The integrated intensity of these features is proportional to the cross-correlation signal. The one-photon features are used for normalization of spectra.

Other approaches we have either tried or considered, such as cross correlation inside a vented vacuum chamber, averaging the cross correlations measured before and after the vacuum chamber, or nonlinear ejection of electrons from the metal walls of the chamber, all suffer from one or more major drawbacks: determination takes place at a location different than the interaction region, which is especially important if the beams are focused; day to day variation in beam position is not corrected for; a fundamentally different process than photoelectron spectroscopy is used, with the potential for unknown time offsets. Among the advantages of ATD: the detachment process is instantaneous for

a bare anion; and halogen anions are usually already present and plentiful in the ion beam, requiring little adjustment to experimental parameters other than laser timing. The ATD signal is moderately weak in comparison to typical FPES signals, but a good quality spectrum can still be collected rapidly since only integrated peak intensities are required, rather than fine details. An example of an ATD photoelectron spectrum of I^- at two time delays (near and far from t_0), is shown in Fig. 25. Other peaks, i.e., one- and two-photon 3ω signals, are independent of time delay and are used to normalize spectra to

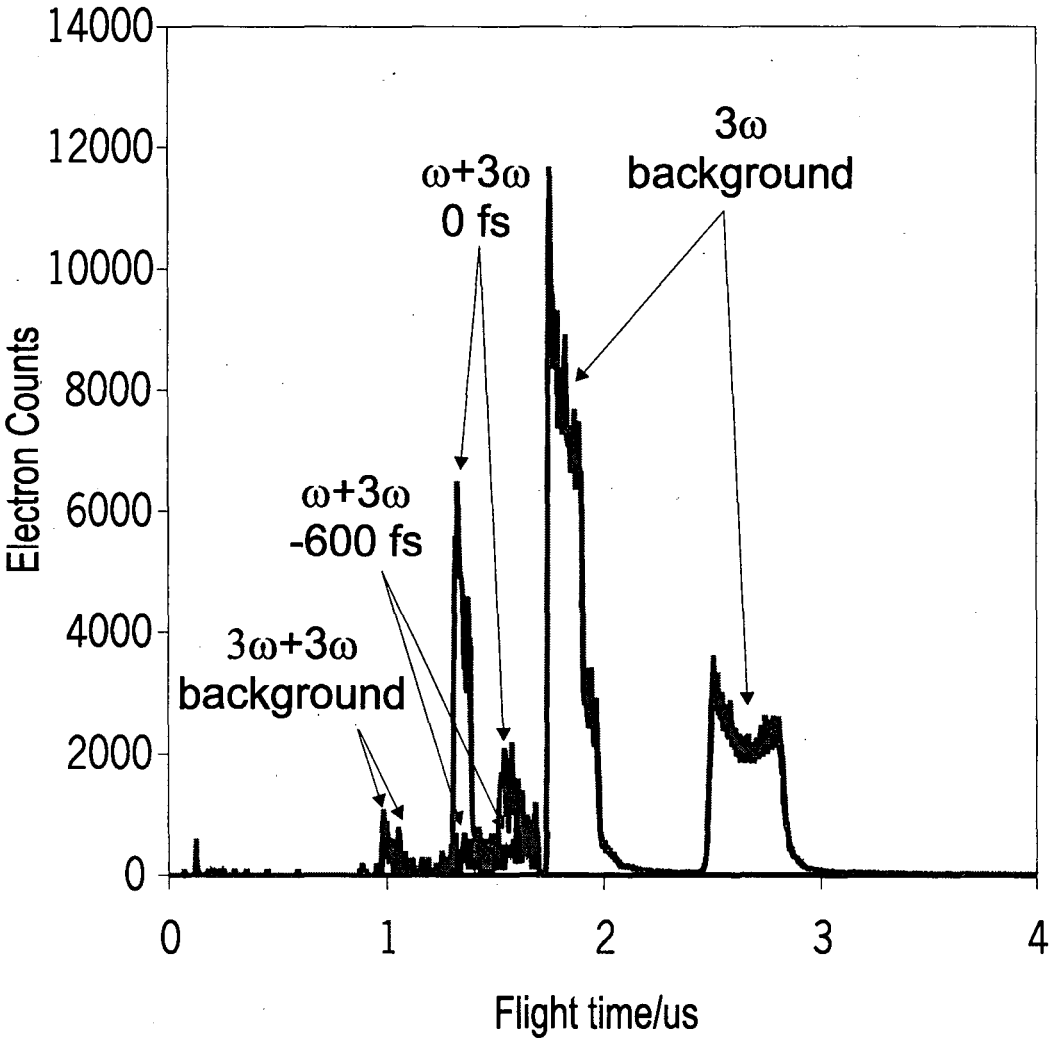


Fig. 25. Example of above-threshold detachment photoelectron spectrum of I^-, showing appearance of additional features when the laser pulses are overlapped.

each other. A plot showing the integrated time-dependent peak intensities *vs.* time, in comparison with a cross correlation spectrum measured on the same day, is shown in Fig. 26. The ATD is considerably broader than the cross correlation (170 vs. 230 fs FWHM).

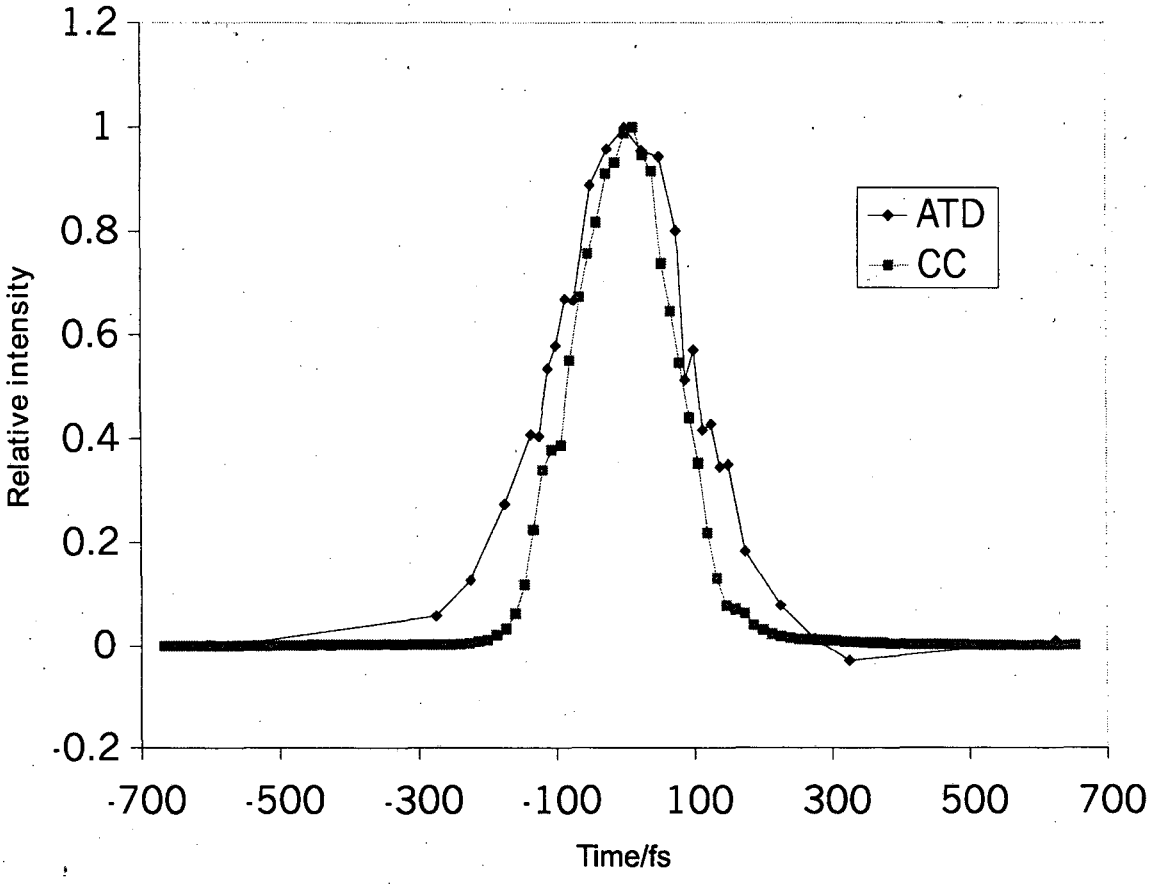


Fig. 26. Integrated ATD peak intensities vs. time, in comparison with cross-correlation of fundamental and third harmonic beams, showing broader pulse overlap inside vacuum.

3.3. Correlation spectra

Auto- and cross-correlation spectra are recorded using a Data Conversion DT2821 analog-to-digital (A/D) card, which plugs into an expansion slot of the lab computer (PC-compatible 486). This flexible device has 12-bit resolution and a software adjustable dynamic range from 1.25 to 10 V. It can read up to eight separate analog signals simultaneously, though only a single one is used, shared between the auto- and cross-

correlation photodiode signals. The A/D conversion process requires approximately 20 μs to complete. For this reason, the signal is connected through 1 MΩ impedance to provide a sufficiently long decay, though doing this also generates a higher voltage level than a smaller impedance connection. A separate trigger signal (TTL) is used to start the digitization, supplied by an output from the Stanford box. It is set 120 μs after the laser trigger to allow sufficient time for the signal to reach its maximum level. Reading the digitized signal requires some assembly language programming, but this has been built into the data acquisition program and is transparent to the user. One item of important practical consequence is that the signal level generally seems to fluctuate a great deal, so that a number of laser shots (~100) must be digitized to obtain a clean signal for each time delay.

4. References

1 W. C. Wiley and I. H. McLaren, Rev. Sci. Instrum. **26,** 1150 (1955).

2 D. R. Cyr, Ph. D. Thesis, University of California, Berkeley (1993).

3 R. Proch and T. Trickl, Rev. Sci. Instrum. **60,** 713 (1989).

4 L.-S. Wang, H.-S. Cheng, and J. Fan, J. Chem. Phys. **102,** 9480 (1995).

5 O. Cheshnovsky, S. H. Yang, C. L. Pettiette, M. J. Craycraft, and R. E. Smalley, Rev. Sci. Instrum. **58,** 2131 (1987).

6 J. M. Alford, P. E. Williams, D. J. Trevor, and R. E. Smalley, Intl. J. Mass. Spec. Ion Proc. **72,** 33 (1986).

7 A. Weaver, Ph.D. Thesis, University of California, Berkeley (1991).

8 M. L. Alexander, N. E. Levinger, M. A. Johnson, D. Ray, and W. C. Lineberger, J. Chem. Phys. **88,** 6200 (1988).

9 J.-C. Diels and W. Rudolph, *Ultrashort Laser Pulse Phenomena: Fundamentals, Techniques, and Applications on a Femtosecond Time Scale* (Academic Press, San Diego, CA, 1996).

10 Y. R. Shen, *The Principles of Nonlinear Optics* (John Wiley & Sons, New York, 1984).

11 R. L. Fork, C. H. B. Cruz, P. C. Becker, and C. V. Shank, Opt. Lett. **12,** 483 (1987).

12 A. Baltuska, Z. Wei, M. S. Pshenichnikov, and D. A. Wiersma, Opt. Lett. **22,** 102 (1997).

13 M. D. Davidson, B. Broers, H. G. Muller, and H. B. van Linden van den Heuvell, J. Phys. B **25,** 3093 (1992).

Chapter 3. Photodissociation dynamics of the I_2^- anion using femtosecond photoelectron spectroscopy*

The photodissociation dynamics of the I_2^- anion have been studied in real-time using femtosecond photoelectron spectroscopy. In this experiment, I_2^- is excited to a dissociative electronic state with an ultrafast pump pulse, and the photoelectron spectrum of the dissociating anion is measured by photodetachment with a second, ultrafast probe pulse. The variation of the photoelectron spectrum with delay time enables one to monitor the dissociating anion from the initial Franck-Condon region of excitation out to the asymptotic region. Dissociation occurs on a time scale of 100 fs. The results are comparised with quantum mechanical simulations using previously published potential energy curves for I_2^-.

1. Introduction

The successful application of time-resolved techniques to gas phase processes occurring on a femtosecond time scale has been one of the most important developments in chemical dynamics during the last ten years.[1,2] The considerable body of work in this area has provided new insights into the photodissociation and reaction dynamics of molecules and clusters. However, nearly all gas-phase femtosecond experiments performed to date have focused on neutral species. The application of these methods to ions, particularly negative ions, is very appealing. In contrast to neutral species, most potential energy surfaces involving negative ions are poorly characterized. The low number densities typical of gas phase negative ion experiments make it difficult to study

* B. J. Greenblatt, M. T. Zanni and D. M. Neumark, *Chem. Phys. Lett.*, **258**, 523 (1996).

the spectroscopy and dynamics of these species using frequency-resolved techniques, such as absorption spectroscopy, laser-induced fluorescence, or multi-photon ionization, that are applied almost routinely to neutral species. Hence, the photodissociation and reaction dynamics of negative ions represent fertile ground for time-resolved experiments. The desirability and feasibility of performing time-resolved experiments on mass-selected anions has been demonstrated in the pioneering work by Lineberger and co-workers[3-6] on dissociation and caging dynamics in $I_2^-(CO_2)_n$ clusters.

These cluster studies provide the motivation for the work described here, in which the photodissociation dynamics of I_2^- are investigated using femtosecond photoelectron spectroscopy (FPES). This is a relatively new technique which, along with the related technique of femtosecond zero electron kinetic energy spectroscopy,[7,8] has recently been applied to excited state dynamics in neutral molecules.[9,10] The results here represent the first application of FPES to negative ions. FPES is a pump-and-probe experiment involving two femtosecond pulses. In our experiment, the first pulse ($h\nu_1$) electronically excites the I_2^- to a repulsive state, and the second ($h\nu_2$) photodetaches the dissociating molecule to form a photoelectron and either two I atoms or an excited I_2 molecule. The overall process is given by:

$$I_2^- \xrightarrow[h\nu_1]{} I_2^{-*} \xrightarrow[\Delta t]{} I^- \cdot\cdot I \xrightarrow[h\nu_2]{} I_2 + e^-, I + I + e^- . \qquad (1)$$

By measuring the photoelectron kinetic energy spectrum as a function of delay time Δt, one can monitor the dissociation dynamics of the electronically excited I_2^- all the way from the Franck-Condon region to the dissociation asymptote.

The I_2^- anion was chosen for these first studies because of its experimental accessibility, and because it is a fundamentally important anion in gas phase and solution

phase chemistry. Chen and Wentworth[11] constructed a set of potential energy curves for the ground and excited states of I_2^- based on Raman spectroscopy in a rare gas matrix, electronic spectroscopy in a crystal, and gas phase dissociative attachment experiments. However, questions remain concerning the accuracy of these curves. For example, the I_2^- electronic spectrum clearly depends on the environment of the ion; the bands in rare gas matrices are shifted by 0.16-0.27 eV to the blue of the bands in a crystalline environment,[12-14] and one expects the gas phase spectrum to differ from either condensed phase spectrum. Recent dissociative attachment results[15] also suggest that the I_2^- potential energy curves in Ref. [11] need to be modified. These curves have been used to simulate time-resolved dynamics of I_2^- in clusters[16] and in various solvents,[17,18] so it is important that they be as accurate as possible. The results presented here provide a stringent test of the available potential energy curves for I_2^-.

Fig. 1 shows the potential energy curves involved in our experiments. I_2^- is excited from its ground $\tilde{X}\ ^2\Sigma_u^+$ state to the low-lying $^2\Pi_{g,1/2}$ excited state by the pump pulse with photon energy $h\nu_1$. The time-dependence of the resulting wavepacket is monitored by measuring its photoelectron spectrum. The photodetaching probe pulse, $h\nu_2$, has sufficient energy to access the $\tilde{X}\ ^1\Sigma_g^+$, $\tilde{A}'\ ^3\Pi_{2u}$, $\tilde{A}\ ^3\Pi_{1u}$, and $\tilde{B}\ ^3\Pi_{0^+u}$ states of I_2. At short delay times, photodetachment will access bound vibrational levels of these I_2 states, but at longer times, when dissociation to I^- + I is complete, one is essentially photodetaching a free I^- ion. Hence, the photoelectron spectrum of the dissociating wavepacket should change substantially with delay time. Since the I_2 states are well characterized,[19-21] the time-resolved photoelectron spectra should serve as a probe of the anion states, as desired.

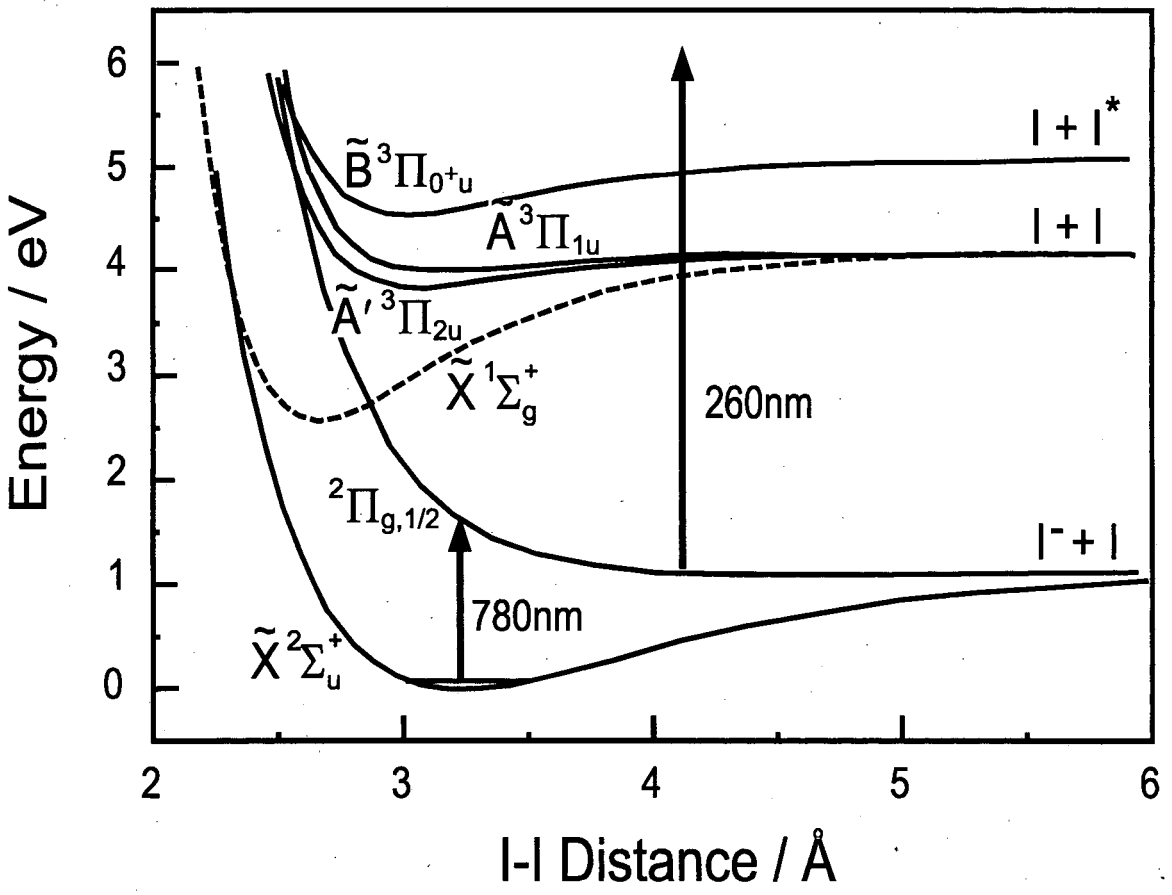


Fig. 1. Potential energy curves for relevant electronic states of I_2^- and I_2 taken from Refs. 11,19-21.

2. Experimental

The experiment consists of two major components: a negative ion photoelectron spectrometer with a "magnetic bottle" electron detector, and a high (1 kHz) repetition rate laser capable of generating sub-100 fs pulses. The photoelectron spectrometer is shown in Fig. 2. It shares several features with spectrometers currently in operation in our laboratory[22] as well as others,[23-25] but is optimized in design so as to be compatible with the laser repetition rate and pulse energy.

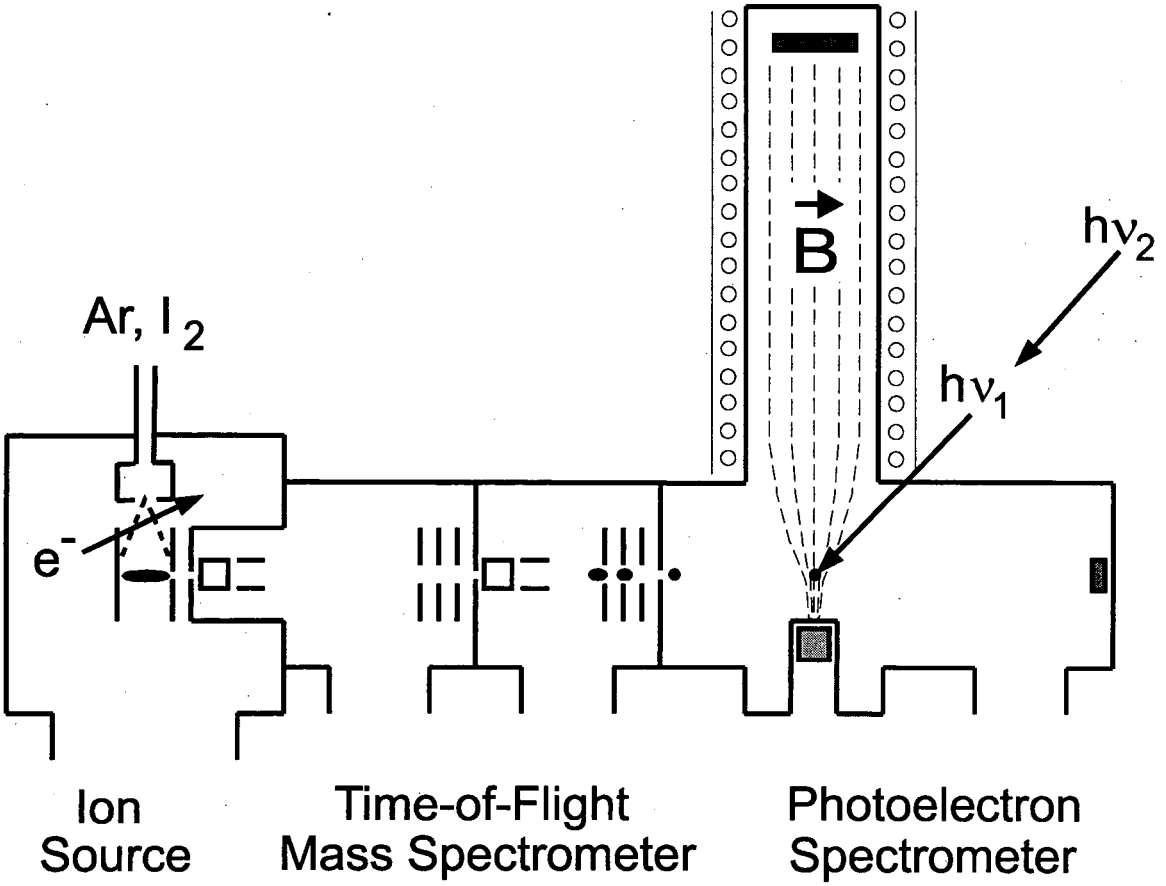


Fig. 2. Schematic of apparatus, showing ion source region, time-of-flight mass spectrometer, and "magnetic bottle" photoelectron time-of-flight spectrometer.

I_2^- anions are generated in a continuous free jet ion source by passing Ar carrier gas at 20 psig over I_2, expanding the resulting mixture through a 100 μm orifice into the source chamber, and crossing the resulting molecular beam just downstream of the orifice with 1 keV electrons from an electron gun. The source chamber is pumped by two Varian VHS-10 diffusion pumps for a total pumping speed of 8800 l/sec. Ions are extracted from the beam and injected into a Wiley-McLaren time-of-flight mass spectrometer[26] by applying pulsed extraction and acceleration fields perpendicular to the molecular beam axis; the final ion beam energy is about 1200 eV. Once the ions are accelerated, they pass through two differentially pumped regions en route to the laser

interaction region. The first differential region is pumped by a Varian VHS-6 diffusion pump. The second differential region and laser interaction region are pumped by Varian V250 turbomolecular pumps, and the base pressure in the latter region is 2×10^{-9} Torr. An in-line microchannel plate detector is used to obtain the time-of-flight mass spectrum of the ion beam.

The ion beam is crossed by the laser pulses 55 cm upstream of the ion detector. A large fraction (>50%) of the resulting photoelectrons is collected using a magnetic bottle time-of-flight photoelectron spectrometer based on the design of Cheshnovsky *et al.*,[25] although we use a strong (0.8 tesla) permanent magnet rather than an electro-magnet to generate the inhomogeneous magnetic field. The electrons are collected at a 75 mm diameter microchannel plate detector 1.4 m from the interaction region, and the arrival time distribution is recorded after each laser shot with a Stanford Research Systems SR430 multichannel scalar. The energy resolution of the spectrometer is currently 150-300 meV, depending on the electron kinetic energy; this will be improved in the near future by pulsed deceleration of the ion beam.[24,25]

The pump and probe laser pulses are generated from a commercial femtosecond laser system. A Coherent Innova-90 Ar^+ laser pumps a Clark-MXR NJA-5 Ti:sapphire oscillator. Selected pulses are amplified using a Clark-MXR regenerative amplifier system that includes a pulse stretcher, a Ti:sapphire regenerative amplifier pumped by a Nd:YAG laser running at a repetition rate of 1 kHz, and a pulse compressor. At 780 nm, the pump pulse wavelength, the pulse width and energy are 85 fs and 800 μJ, respectively. About 70% of this beam is directed into a frequency tripling unit (CSK Optronics 8315A), resulting in a probe pulse at 260 nm with width and energy of 110 fs

and 18 μJ, respectively. (The pulse width of the probe pulse is measured by cross-correlation with the pump pulse using a KDP crystal for frequency differencing.) The remainder of the 780 nm pulse passes through a variable delay line and is then collinearly recombined with the probe pulse prior to entering the vacuum chamber.

The UV probe pulse spreads when it passes through the vacuum chamber window, and this window also affects the delay between the pump and probe pulses. Two-color above threshold detachment (ATD) of I^- is used to characterize the laser pulses inside the vacuum chamber.[27] The probe pulse alone produces the characteristic photoelectron spectrum of I^- (see below). When the pump and probe pulses temporally overlap, additional peaks are observed that correspond to shifting the I^- spectrum by 1.6 eV towards higher electron kinetic energy; this is the photon energy of the pump pulse. From the intensity of this two-color signal as a function of pump-probe delay, we determine the zero-delay time and the cross-correlation of the pump and probe pulses inside the vacuum chamber. This yields a pulse width of 140 fs for the probe pulse.

The combination of high laser repetition rate and high electron collection efficiency results in rapid data collection. In the data set presented below, only 50 sec of data collection are required at each time delay. The spectra are not background-free because one probe photon can photodetach ground state I_2^-. At each time delay, background subtraction is accomplished by collecting 50% of the data (i.e. for 25 sec) at long negative delays (-2 ps), where the probe pulse fires well before the pump pulse. This background is suitably scaled and subtracted from the raw spectra to yield the spectra in the following section.

3. Results and Analysis

Experimental results are shown in Fig. 3. In the top half of Fig. 3, three photoelectron spectra are superimposed, at Δt = 0, 150, and 425 fs. The spectrum at the longest delay time, Δt = 425 fs, is essentially the I^- photoelectron spectrum; the two peaks centered at 0.75 and 1.7 eV represent transitions to the $I(^2P_{3/2})$ and $I^*(^2P_{1/2})$ states, respectively. A comparison with the two spectra at shorter delay times shows that the intensities of these atomic transitions increase monotonically with Δt. In addition, there is a transient signal on the high electron kinetic energy side of each atomic transition that is of comparable intensity in the spectra at Δt = 0 and 150 fs but has decayed to zero by Δt = 425 fs. A comparison of the spectra at Δt = 0 and 150 fs shows that the transient signal is shifted towards the atomic transitions at the longer delay time.

The full data set of 21 photoelectron spectra is shown as a three-dimensional surface plot in the bottom half of Fig. 3. This plot emphasizes the temporal structure of the signal at each electron kinetic energy, and shows that depending on the electron kinetic energy, the signal is either monotonically increasing or transient with a full width at half-maximum (FWHM) of about 200 fs. The value of Δt at which the transient signal reaches a maximum depends on the electron kinetic energy, as indicated by the dark lines in Fig. 3. As the electron kinetic energy decreases (i.e. moves towards the atomic transition), the maximum occurs at longer values of Δt.

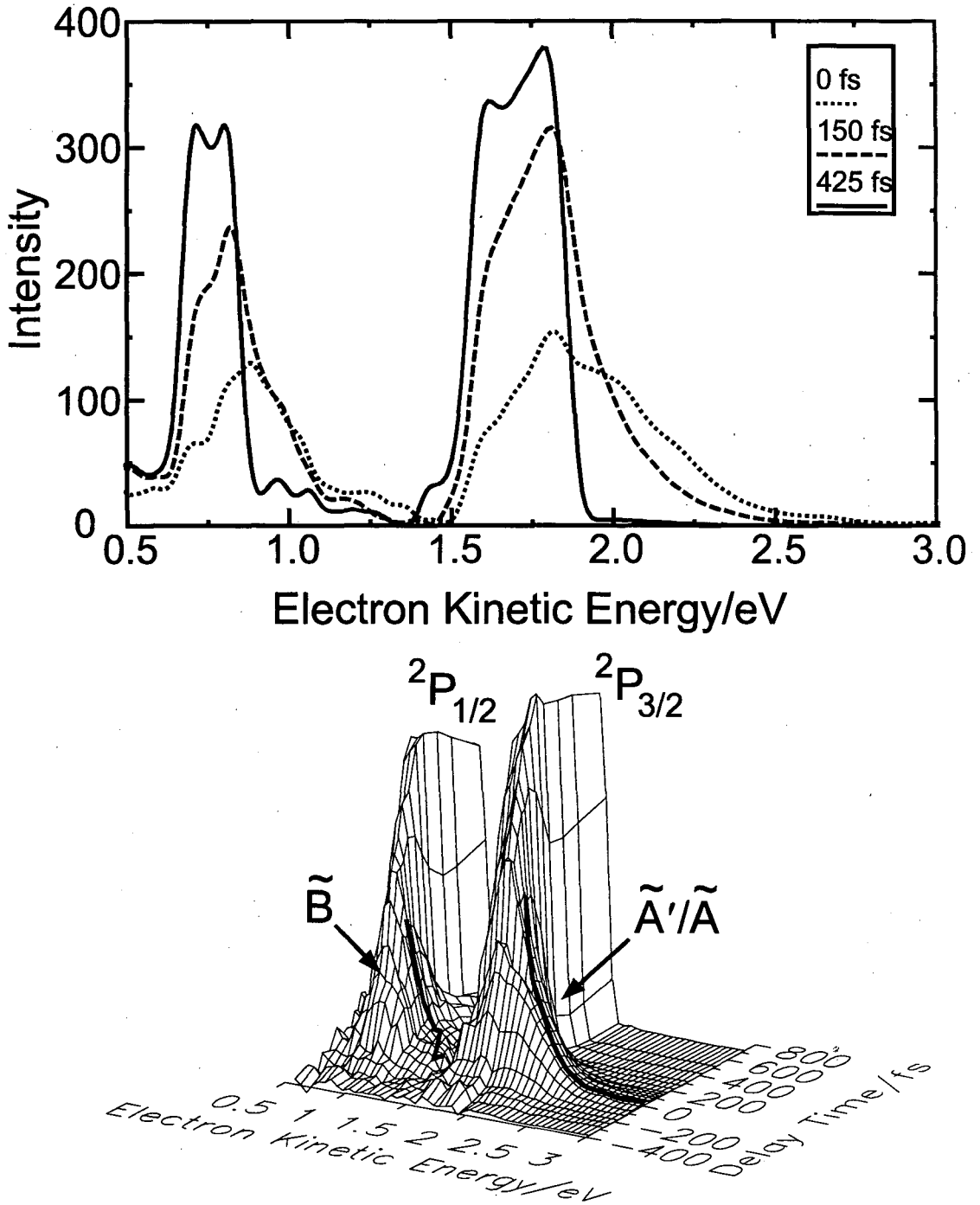


Fig. 3. Experimental femtosecond photoelectron spectra of I_2^-. Upper panel: spectra at three delay times. Lower panel: spectra at 21 delay times ranging from -400 to 725 fs. Assignments of various energy ranges are indicated. The dark lines show the delay time at which the maximum intensity of transient signal occurs for each electron kinetic energy.

The monotonically increasing I^- signal clearly comes from fully dissociated I_2^-. The transient signal can be assigned with reference to the potential energy curves in Fig. 1. At 260 nm, the $\tilde{X}\,^1\Sigma_g^+$, $\tilde{A}'\,^3\Pi_{2u}$, $\tilde{A}\,^3\Pi_{1u}$, and $\tilde{B}\,^3\Pi_{0^+u}$ states of I_2 should be energetically accessible by photodetachment from the excited $I_2^-\,^2\Pi_{g,1/2}$ state over the full range of internuclear distances probed in the experiment. However, the $\tilde{X}\,^1\Sigma_g^+$ state of I_2 cannot be reached by a one-electron photodetachment transition from the $^2\Pi_{g,1/2}$ state of I_2^-; the valence molecular orbital configurations for these states are $\sigma_g^2\,\pi_u^4\,\pi_g^4$ and $\sigma_g^2\,\pi_u^4\,\pi_g^3\,\sigma_u^2$, respectively. Transitions from the excited anion to the $\tilde{X}\,^1\Sigma_g^+$ state should therefore be weak or non-existent, whereas the other three states are accessible by one-electron photodetachment transitions. Fig. 1 shows that the excited anion and neutral potential energy curves are closer at short internuclear distances than in the asymptotic region. We therefore assign the transient signal on the high kinetic energy side of the $I^-\rightarrow I(^2P_{3/2})$ and $I^-\rightarrow I^*(^2P_{1/2})$ peaks to transitions from the dissociating anion to the $\tilde{A}'\,^3\Pi_{2u}$ / $\tilde{A}\,^3\Pi_{1u}$ states and the $\tilde{B}\,^3\Pi_{0^+u}$ state, respectively; these assignments are indicated in Fig. 3.

In order to interpret the spectra in more detail, quantum mechanical simulations of the time-resolved photoelectron spectra have been performed, using a wavepacket propagation scheme developed by Kosloff.[28] Wavefunctions $|\psi_n(t)\rangle$, with n = 1, 2 or 3, are represented on a spatial grid for each of three potential energy curves: 1 = I_2^- ($\tilde{X}\,^2\Sigma_u^+$); 2 = I_2^- ($^2\Pi_{g,1/2}$); 3 = I_2 ($\tilde{A}'\,^3\Pi_{2u}$, $\tilde{A}\,^3\Pi_{1u}$, or $\tilde{B}\,^3\Pi_{0^+u}$). Morse functions[11,19-21] were used for all states.

The simulations are carried out in two steps. The wavepacket for the dissociating anion, $|\psi_2(t)\rangle$, is found by numerically integrating the time-dependent Schrödinger equation

$$i\hbar\frac{d}{dt}\begin{pmatrix}|\psi_1(t)\rangle\\|\psi_2(t)\rangle\end{pmatrix}=\begin{pmatrix}H_1 & -\mu_{12}E_{12}^*(t)\\-\mu_{12}E_{12}(t) & H_2\end{pmatrix}\begin{pmatrix}|\psi_1(t)\rangle\\|\psi_2(t)\rangle\end{pmatrix} \tag{2}$$

Here H_n is the nuclear Hamiltonian for state n, $E_{12}(t) = E_{12}\text{sech}(t/T_{12})\exp(-i\omega_{12}t)$ is the time-dependent pump laser field (E_{12} is the maximum field intensity; T_{12}, the pulse width; ω_{12}, the carrier frequency), and μ_{12} is the transition dipole moment between states 1 and 2, assumed to be constant for all internuclear distances.

First order perturbation theory is then used to calculate $|\psi_3(t;\varepsilon,\Delta t)\rangle$, the neutral vibrational wavefunctions corresponding to electron kinetic energy ε.[29] This is given by:

$$|\psi_3(t;\varepsilon,\Delta t)\rangle = -\frac{i\mu_{23}}{\hbar}\int_{-\infty}^{t} dt' e^{-i(H_3+\varepsilon)(t-t')/\hbar}E_{23}(t'-\Delta t)|\psi_2(t')\rangle \tag{3}$$

where Δt is the time delay between pump and probe pulses, H_3 is the nuclear Hamiltonian for state 3, ε is the electron kinetic energy, and $E_{23}(t - \Delta t)$ is the probe laser field, with the same assumed functional form as for $E_{12}(t)$. The transition dipole moment μ_{23} is again assumed to be constant for all distances. The time-dependent photoelectron intensity is then obtained by calculating the norm of $|\psi_3\rangle$ in the long-time limit:[29,30]

$$P(\varepsilon,\Delta t)=\lim_{t\to\infty}\langle\psi_3(t;\varepsilon,\Delta t)|\psi_3(t;\varepsilon,\Delta t)\rangle=\frac{\mu_{23}^2}{\hbar^2}\left|\int_{-\infty}^{\infty}dt' e^{i\varepsilon t'/\hbar}\left[E_{23}(t'-\Delta t)e^{iH_3t'/\hbar}|\psi_2(t')\rangle\right]\right|^2 \tag{4}$$

Note that the bracketed expression in the integrand is the argument of a Fourier transform. Thus, once the set of wavefunctions $|\chi(t')\rangle = e^{iH_3t'/\hbar}|\psi_2(t')\rangle$ is determined, the entire photoelectron spectrum is readily calculated. In addition, since $E_{23}(t' - \Delta t)$ is a

scalar multiplier, it can be applied independently of $|\chi(t')\rangle$, allowing calculation of the spectrum for arbitrary Δt or probe pulse shape without re-determining $|\psi_2(t')\rangle$.

Raw spectra were convoluted with the instrument resolution function for an isotropic electron angular distribution, assuming electrons are collected over 4π steradians in our experiment:

$$p(E,\varepsilon) = \sqrt{1 - \frac{M}{4m_e U\varepsilon}\left(E - \varepsilon - \frac{m_e U}{M}\right)^2} \tag{5}$$

Here M = ion mass, m_e = electron mass, U = ion beam energy, ε = electron kinetic energy (center-of-mass frame) and E = electron kinetic energy (lab frame). Using M = 254 amu and U = 1200 eV gives an energy resolution of 0.20 eV for 1 eV electrons, in good agreement with experiment.

The spectra arising from transitions to the three neutral states of I_2 are calculated separately, then summed using the following weighting criterion: μ_{23} is assumed equal for transitions to the $\tilde{A}'\,^3\Pi_{2u}$ and $\tilde{A}\,^3\Pi_{1u}$ states, and μ_{23} for the transition to the $\tilde{B}\,^3\Pi_{0^+u}$ state is adjusted so that the ratio of $I^*(^2P_{1/2})$ to $I(^2P_{3/2})$ intensities (at large delay time) reproduces the experimental value of 0.9.

The simulated spectra are shown in Fig. 4. Overall, the experimental and simulated spectra are in reasonable agreement. The transient signal appears over the same energy range in both the experimental and simulated spectra, indicating that our assignment of the transient features discussed above is correct. However, there clearly are differences between the two spectra, and these are discussed in the next section.

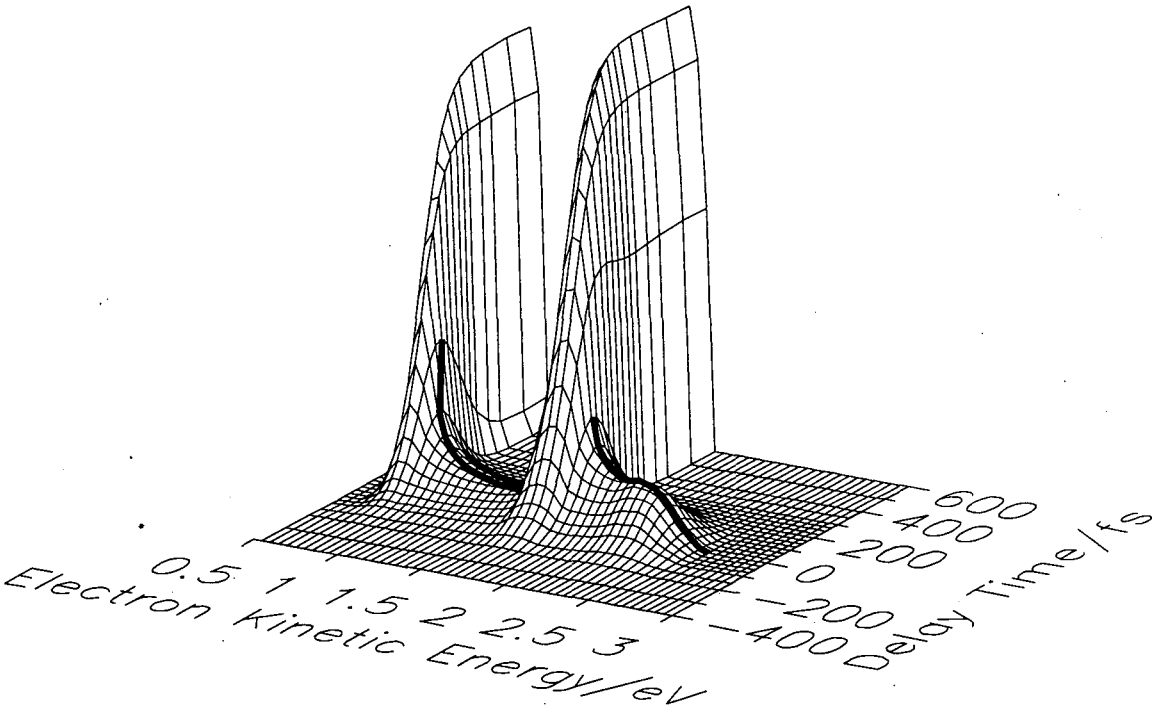


Fig. 4. Simulated femtosecond photoelectron spectra of I_2^- using Eq. 4 in text. Delay times range from -400 to 600 fs. The dark lines show the delay time at which the maximum intensity of transient signal occurs for each electron kinetic energy.

4. Discussion

From the experimental spectra alone, one can obtain an approximate time scale for dissociation of excited I_2^- from the rise time of the signal corresponding to the product atomic transitions. This is plotted in Fig. 5 for electron kinetic energies of 1.65 eV and 0.75 eV, corresponding to the $I^- \rightarrow I(^2P_{3/2})$ and $I^- \rightarrow I(^2P_{1/2})$ transitions, respectively. In both cases, the electron signal reaches 50% of its maximum value by Δt = 140 fs. A more detailed picture of the dynamics comes from the temporal profiles at constant electron kinetic energy. As the electron kinetic energy decreases from the onset of the $\tilde{A}'/\tilde{A}$ transient at 2.6 eV to the start of the $I^- \rightarrow I(^2P_{3/2})$ transition at 1.9 eV, the maximum in the temporal profile (dark line, Fig. 3) increases from Δt = 10 to Δt = 110 fs. A similar shift is seen for the B transient. This shift essentially tracks the dissociating wavepacket from

the initial Franck-Condon region of excitation at short times, where the vertical detachment energy from the anion $^2\Pi_{g,1/2}$ state is smaller (see Fig. 1), to the asymptotic region at longer times where the vertical detachment energy is larger.

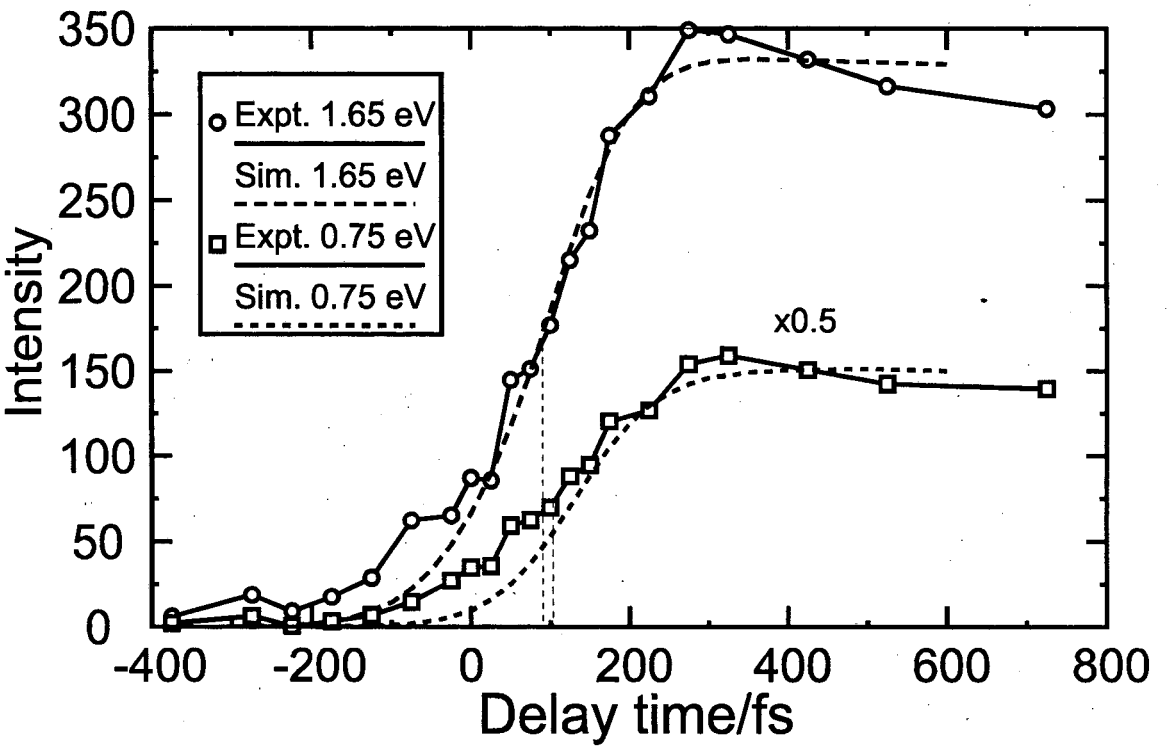


Fig. 5. Appearance of signal versus delay time at electron kinetic energies of 1.65 and 0.75 eV, corresponding to I^- product. Solid lines: experimental spectra. Dashed lines: simulated spectra. Vertical line indicates Δt = 100 fs.

We next compare the experimental and simulated spectra. The transient signals are noticeably less intense in the simulated spectra, and there is a broad peak centered at 2.5 eV and Δt = 0 fs in the simulated spectra that is not apparent in the experimental spectra. However, the overall timescales in the simulated spectra are similar to those in the experiment. Fig. 5 shows that the 50% level of the simulated signal at 1.65 eV occurs at 100 fs, with a slighly longer rise time (120 fs) at 0.75 eV. From Fig. 4, the maxima in the temporal profiles shift by 100 fs in the energy range of the two transients.

The lower intensity in the simulated spectra may simply result from our assumption that the transition dipole for photodetachment, μ_{23}, is constant in Eq. 3; this discrepancy would be resolved if μ_{23} were larger for small internuclear distances, before dissociation is complete. On the other hand, it appears that the potential energy curves used in the simulations reproduce the main features of the experimental dynamics reasonably well. A new set of I_2^- potential energy curves has just been published, [31] and it will be of interest to simulate the spectra using these new curves and compare the results to experiment.

5. Summary

This Letter represents the first application of femtosecond photoelectron spectroscopy to negative ions, specifically the photodissociation dynamics of I_2^-. This method offers considerable promise for performing time-resolved studies of molecular and cluster anions. The general advantage afforded by FPES is that the probe pulse need not be tunable; the photoelectron spectrum maps out the dissociating anion state onto all neutral states that are energetically accessible at the photon energy of the probe laser. In the case of I_2^-, the wavefunction for the dissociating anion is simultaneously mapped onto the $\tilde{A}'^3\Pi_{2u}$, $\tilde{A}^3\Pi_{1u}$, and $\tilde{B}^3\Pi_{0^+u}$ states of I_2. Moreover, since electron binding energies in negative ions are relatively low, only one photon is typically required to photodetach the dissociating ion anywhere along the reaction coordinate. One can therefore analyze the spectra relatively easily, in contrast to analogous experiments on neutrals where multiple photon absorption is typically required for ionization.

6. Acknowledgments

This research is supported by the National Science Foundation under Grant No. CHE-9404735. Support from the Defense University Research Instrumentation Program and Air Force Office of Scientific Research under Grant No. F49620-95-1-0078 is also gratefully acknowledged. The authors thank Prof. Yongqin Chen for many invaluable discussions.

7. References

1 A. H. Zewail, J. Phys. Chem. **97,** 12427 (1993).

2 J. C. Polanyi and A. H. Zewail, Acc. Chem. Res. **28,** 119 (1995).

3 D. Ray, N. E. Levinger, J. M. Papanikolas, and W. C. Lineberger, J. Chem. Phys. **91,** 6533 (1989).

4 J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, H. K. Buchenau, and W. C. Lineberger, J. Chem. Phys. **99,** 8733 (1993).

5 J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, J. R. Gord, and W. C. Lineberger, J. Chem. Phys. **97,** 7002 (1992).

6 J. M. Papanikolas, J. R. Gord, N. E. Levinger, D. Ray, V. Vorsa, and W. C. Lineberger, J. Phys. Chem. **95,** 8028 (1991).

7 T. Baumert, R. Thalweiser, and G. Gerber, Chem. Phys. Lett. **209,** 29 (1993).

8 I. Fischer, D. M. Villeneuve, M. J. J. Vrakking, and A. Stolow, J. Chem. Phys. **102,** 5566 (1995).

9 B. Kim, C. P. Schick, and P. M. Weber, J. Chem. Phys. **103,** 6903 (1995).

10 D. R. Cyr and C. C. Hayden, J. Chem. Phys. **104,** 771 (1996).

11 E. C. M. Chen and W. E. Wentworth, J. Phys. Chem. **89,** 4099 (1985).

12 C. J. Delbecq, W. Hayes, and P. H. Yuster, Phys. Rev. **121,** 1043 (1961).

13 L. Andrews, J. Am. Chem. Soc. **98,** 2152 (1976).

14 H. N. Hersh, J. Chem. Phys. **31,** 909 (1959).

15 R. Azria, R. Abouaf, and D. Teillet-Billy, J. Phys. B: At. Mol. Opt. Phys. **21,** L213 (1988).

16 J. M. Papanikolas, P. E. Maslen, and R. Parson, J. Chem. Phys. **102,** 2452 (1995).

17 A. E. Johnson, N. E. Levinger, and P. F. Barbara, J. Phys. Chem. **96,** 7841 (1992).

18 P. K. Walhout, J. C. Alfano, K. A. M. Thakur, and P. F. Barbara, J. Phys. Chem. **99,** 7568 (1995).

19 X. N. Zheng, S. L. Fei, M. C. Heaven, and J. Tellinghuisen, J. Chem. Phys. **96,** 4877 (1992).

20 X. N. Zheng, S. L. Fei, M. C. Heaven, and J. Tellinghuisen, J. Molec. Spectrosc. **149,** 399 (1991).

21 J. I. Steinfeld, R. N. Zare, L. Jones, M. Lesk, and W. Klemperer, J. Chem. Phys. **42,** 25 (1965).

22 R. B. Metz, A. Weaver, S. E. Bradforth, T. N. Kitsopoulos, and D. M. Neumark, J. Phys. Chem. **94,** 1377 (1990).

23 L. A. Posey, M. J. DeLuca, and M. A. Johnson, Chem. Phys. Lett. **131,** 170 (1986).

24 C.-Y. Cha, G. Gantefor, and W. Eberhardt, Rev. Sci. Instrum. **63,** 5661 (1992).

25 O. Cheshnovsky, S. H. Yang, C. L. Pettiette, M. J. Craycraft, and R. E. Smalley, Rev. Sci. Instrum. **58,** 2131 (1987).

26 W. C. Wiley and I. H. McLaren, Rev. Sci. Instrum. **26,** 1150 (1955).

27 M. D. Davidson, B. Broers, H. G. Muller, and H. B. van Linden van den Heuvell, J. Phys. B **25,** 3093 (1992).

28 R. Kosloff, Annu. Rev. Phys. Chem. **45,** 145 (1994).

29 C. Meier and V. Engel, Phys. Rev. Lett. **73,** 3207 (1994).

30 M. Seel and W. Domcke, J. Phys. Chem. **95,** 7806 (1991).

31 J. G. Dojahn, E. C. M. Chen, and W. E. Wentworth, J. Phys. Chem. **100,** 9649 (1996).

Chapter 4. Time-resolved photodissociation dynamics of $I_2^-(Ar)_n$ clusters using anion femtosecond photoelectron spectroscopy*

Anion femtosecond photoelectron spectroscopy (FPES) is used to follow the dynamics of the $I_2^-(Ar)_6$ and $I_2^-(Ar)_{20}$ clusters subsequent to photodissociation of the I_2^- chromophore. The experiments show that photodissociation of the I_2^- moiety in $I_2^-(Ar)_6$ is complete by ~200 femtoseconds (fs), just as in bare I_2^-, but also that attractive interactions between the departing anion fragment and the solvent atoms persist for 1200 femtoseconds. Photodissociation of $I_2^-(Ar)_{20}$ results in caging of the I_2^- followed by recombination and vibrational relaxation on the excited $\tilde{A}\ ^2\Pi_{g,3/2}$ state and the ground $\tilde{X}\ ^2\Sigma_u^+$ states; these processes are complete in 35 picoseconds (ps) and 200 picoseconds, respectively.

1. Introduction

Our understanding of the potential energy surfaces governing the dynamics of elementary chemical reactions in the gas phase has grown significantly during the past 10 years, largely because of the development of new frequency and time-resolved experimental techniques[1,2] combined with theoretical advances in quantum chemistry[3] and reaction dynamics[4]. A very appealing new direction in this field is to investigate, in a systematic way, the effects of solvation on reaction dynamics. Studies of chemical reactions occurring within size-selected clusters provides an elegant means of achieving this goal, because one can monitor qualitative changes that occur as a function of cluster

---

* B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, *Science* **276**, 1675 (1997).

size and ultimately learn how the dynamics of an elementary unimolecular or bimolecular reaction evolve as one approaches a condensed phase environment.[5] It is particularly useful to perform such experiments on ionic clusters, for which size-selection is straightforward. We recently performed a time-resolved study of the photodissociation dynamics of the I_2^- anion using a new technique, anion femtosecond photoelectron spectroscopy (FPES)[6]. We now apply this method to follow the dynamics that result from photodissociation of the I_2^- chromophore in the clusters $I_2^-(Ar)_6$ and $I_2^-(Ar)_{20}$. These experiments yield time-resolved measurements of the anion-solvent interactions subsequent to photodissociation and, in the case of $I_2^-(Ar)_{20}$, provide new insight into the caging and recombination dynamics of the I_2^- moiety.

Anion FPES is a pump-probe experiment that uses two femtosecond pulses, a pump pulse that photodissociates an anion (or anion chromophore in a cluster) and a probe pulse that ejects an electron from the dissociating species. By measuring the resulting PE spectrum at various delay times, the experiment yields "snapshots" of the dissociation dynamics, and in particular probes how the local environment of the excess electron evolves with time. This highly multiplexed experiment yields information on the entire photoexcited wavepacket at each delay time, in contrast to most pump-and-probe experiments in which signal is observed only if there is an absorption at the frequency of the probe pulse. Although FPES has also been applied to neutrals,[7-9] the anion experiment is inherently mass-selective, making it especially useful in studies of size-selected clusters.

The present work builds on the experiments of Lineberger and co-workers,[10-12] who performed one-photon photodissociation and time-resolved pump-and-probe

experiments on size-selected $I_2^-(CO_2)_n$ and $I_2^-(Ar)_n$ cluster anions, and on the time-resolved studies of neutral $I_2(Ar)_n$ clusters by Zewail and co-workers.[13] The one-photon cluster anion experiments yield the asymptotic daughter ion distributions as a function of initial cluster size, in particular the relative amounts of "caged" $I_2^-(Ar)_{m1<n}$ products, in which the I and I^- photofragments are trapped by the solvent atoms and eventually recombine, versus "uncaged" $I^-(Ar)_{m2<n}$ products in which trapping does not occur. Only uncaged products are observed from the photodissociation of $I_2^-(Ar)_6$, with ArI^- as the dominant product, indicating that there are not enough solvent atoms to trap the recoiling photofragments. In contrast, the solvent shell is approximately complete for $I_2^-(Ar)_{20}$ so that only caged products are seen. Moreover, photodissociation of $I_2^-(Ar)_{20}$ results in two distinct recombination channels formed with approximately equal yield: bare I_2^- and $I_2^-(Ar)_n$ with $\langle n \rangle = 11$. The time-resolved experiments on $I_2^-(Ar)_{20}$ yield a time constant of 127 ps for recovery of the I_2^- absorption[12]; this represents the overall time scale for recombination and relaxation of the I_2^- product. Our experiment provides a more complete picture of the dynamics following excitation of $I_2^-(Ar)_6$ and $I_2^-(Ar)_{20}$, and in particular clarifies the origin of the two product channels seen for $I_2^-(Ar)_{20}$.

2. Experimental

The FPES experiment is described in detail elsewhere.[6] Briefly, a pulsed, mass-selected beam of cold cluster anions is intercepted by the pump and probe pulses at the focus of a "magnetic bottle" time-of-flight PE spectrometer. The two laser pulses are generated by a Ti:sapphire oscillator/regenerative amplifier system (Clark MXR) operating at a repetition rate of 500 Hz. The pump pulse at 780 nm and the probe pulse at

260 nm are 80 and 100 fs long, respectively. The high laser repetition rate combined with the high (>50%) collection efficiency of the magnetic bottle analyzer results in rapid data acquisition; each spectrum is typically obtained in 40 to 80 s for I_2^- and $I_2^-(Ar)_6$, and 10 to 15 minutes for $I_2^-(Ar)_{20}$. At the ion beam energy used in this work (1750 eV), the electron energy resolution of the spectrometer at 1 eV electron kinetic energy (eKE) is 0.25 eV for I_2^-, 0.18 eV for $I_2^-(Ar)_6$, and 0.12 eV for $I_2^-(Ar)_{20}$.

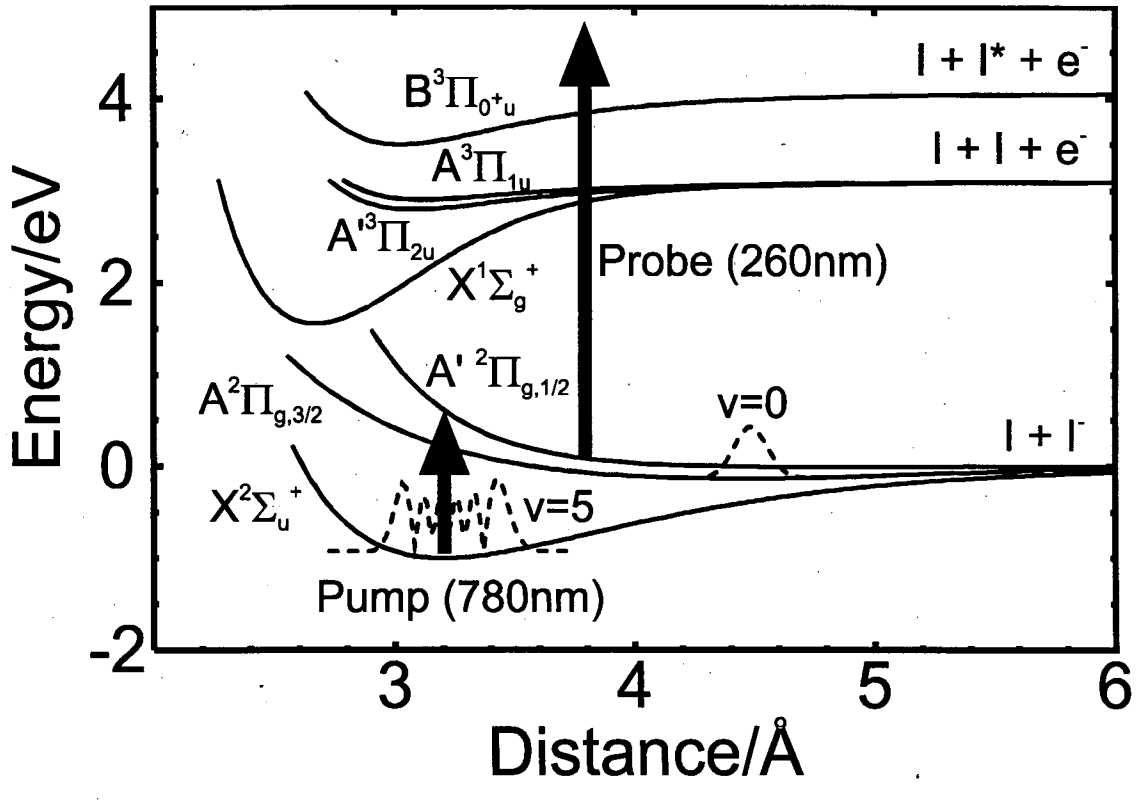


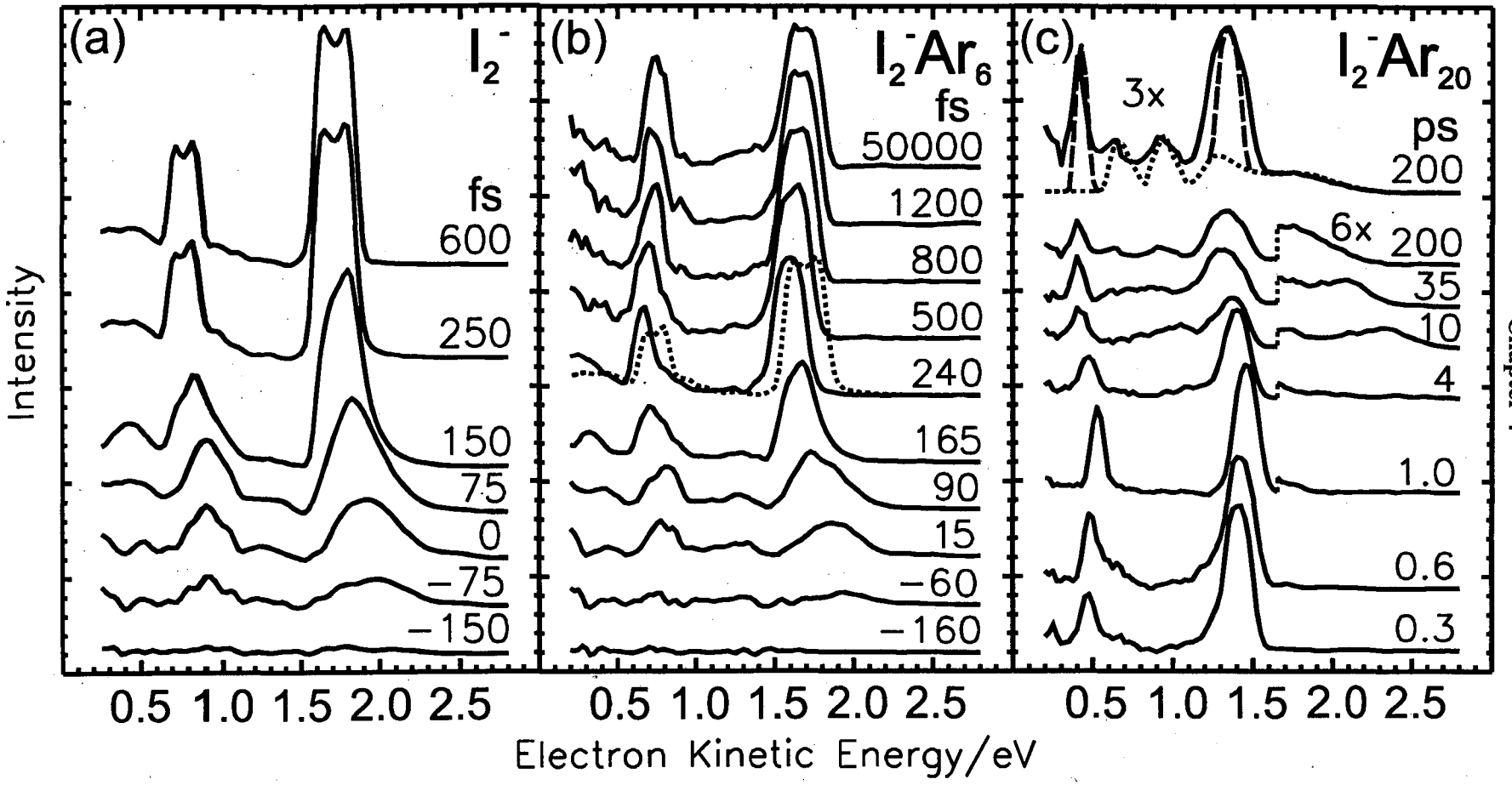
Fig. 1. Potential energy curves for the low-lying electronic states of I_2^- and I_2. The curves for the I_2^- $\tilde{X}\ ^2\Sigma_u^+$ $\tilde{A}'\ ^2\Pi_{g,1/2}$ states are taken from Refs. [14] and [15], respectively. For the $\tilde{X}$ state, D_e=1.01 eV and R_e=3.205 Å. The $\tilde{A}\ ^2\Pi_{g,3/2}$ state is described in the text. The v=0 and v=5 wavefunctions on the I_2^- $\tilde{X}$ and $\tilde{A}$ states are also shown (see text). The I_2 curves are from Ref. [16] and references therein.

The relevant potential energy curves for I_2^- and I_2 are shown in Fig. 1.[14-16] The pump pulse excites the $\tilde{A}'\ ^2\Pi_{g,1/2} \leftarrow \tilde{X}\ ^2\Sigma_u^+$ transition in I_2^-, creating a localized wavepacket on the repulsive excited state. The probe pulse detaches the dissociating ion to the various low-lying states of I_2 shown in Fig. 1. In the bare ion, rapid and direct dissociation to $I^- + I(^2P_{3/2})$ occurs. However, when the same I_2^- transition is excited in an $I_2^-(Ar)_n$ cluster, the recoiling fragments interact with the Ar atoms, and the FPES experiment provides a sensitive probe of these interactions.

3. Results

Fig. 2 shows selected PE spectra at various pump-probe delay times τ for I_2^-, $I_2^-(Ar)_6$ and $I_2^-(Ar)_{20}$. The spectra of I_2^- in Fig. 2a have been discussed in detail previously.[6] Two peaks centered at eKE = 1.70 eV and 0.75 eV rise monotonically with increasing delay; these correspond to photodetachment of the I^- photodissociation product to the $I(^2P_{3/2})$ and $I^*(^2P_{1/2})$ atomic states, respectively. In addition, there is a transient signal peaking at τ=50-100 fs on the high electron energy side of each product peak attributed to the dissociating ions. The product peaks dominate the spectra by 200 fs, and no further changes are observed at later times, indicating dissociation is complete.

Fig. 2 (on next page). Anion femtosecond photoelectron spectra at various pump-probe delay times τ for (a) I_2^-, (b) $I_2^-(Ar)_6$, and (c) $I_2^-(Ar)_{20}$. In Fig. 2b, the I_2^- spectrum at 250 fs is superimposed on the $I_2^-(Ar)_6$ spectrum at 240 fs for comparison. In Fig 2c, the signal at eKE > 1.6 eV is magnified by a factor of 6 for $\tau \geq 1$ ps. A simulation of the $I_2^-(Ar)_{20}$ spectrum at 200 ps is superimposed on the experimental spectrum at the top of Fig. 2c. Contributions to the simulation from transitions originated from the I_2^- $\tilde{X}$ state (···) and $\tilde{A}$ state (---) are indicated.


(a)
I_2^-
fs
600
250
150
75
0
−75
−150
(b)
$I_2^-Ar_6$
fs
50000
1200
800
500
240
165
90
15
−60
−160
(c)
$I_2^-Ar_{20}$
3x
ps
200
6x
200
35
10
4
1.0
0.6
0.3
Intensity
0.5 1.0 1.5 2.0 2.5
0.5 1.0 1.5 2.0 2.5
0.5 1.0 1.5 2.0 2.5
Electron Kinetic Energy/eV

The $I_2^-(Ar)_6$ spectra in Fig. 2b resemble the I_2^- spectra at first glance, particularly for $\tau<240$ fs. By $\tau=240$ fs, the spectra for $I_2^-(Ar)_6$ consist of two peaks clearly analogous to the atomic $I \leftarrow I^-$ transitions in the I_2^- spectra. However, these two "I^-" peaks occur at 0.12 lower eKE than for bare I^-. In addition, they gradually shift toward 0.10 eV higher eKE as τ increases from 240 to 1200 fs. The spectra do not change after $\tau=1200$ fs.

The spectra for $I_2^-(Ar)_{20}$ in Fig. 2c show the same general trends up to $\tau=1$ ps. Two new trends are seen at later times, however. First, as τ increases from 1 to 35 ps, the two "I^-" peaks apparently reverse direction and shift toward lower eKE by about 0.14 eV. Second, a new, broad feature at high electron energy (eKE>1.6 eV) appears at $\tau>4$ ps. This feature shifts toward lower eKE until $\tau=200$ ps, after which no further significant evolution occurs. Two small peaks between the two "I^-" peaks also grow in on this time scale.

4. Discussion

The $I_2^-(Ar)_6$ spectra indicate that the dynamics subsequent to photoexcitation can be divided into two time regimes. At early times ($\tau \leq 240$ fs), the I_2^- chromophore dissociates to $I^- + I$. The similarity between these spectra and those for I_2^- indicates that the primary bond-breaking dynamics are not affected by clustering, and that this process is complete by 240 fs. The interpretation of the spectra in the second time regime, from 240 to 1200 fs, is aided by our previous measurements of the electron affinities of $I(Ar)_n$ clusters, which show that each Ar atom increases the electron affinity by ~ 25 meV.[17] The 0.12 eV energy offset in the "I^-" peaks at 240 fs relative to bare I^- is what would be expected for an I^- ion bound to 5 Ar atoms. As τ increases, the shifts of these peaks

toward higher eKE indicate that the I^- is interacting with progressively fewer Ar atoms. The spectrum at 1200 fs is that expected for ArI^-; this is consistent with the mass-resolved experiments[11] that show ArI^- to be the dominant product from $I_2^-(Ar)_6$ photodissociation. Thus, the evolution of the spectra from 240 to 1200 fs reflects the progressively weaker interactions between the solvent atoms and the I^- fragment, with formation of the asymptotic ArI^- product complete by 1200 fs.

One picture of the dynamics during the second time regime consistent with the spectra is that once the I_2^- chromophore is dissociated, the neutral I atom is ejected, leaving behind a vibrationally hot $I^-(Ar)_n$ cluster from which Ar atoms evaporate until the available energy is dissipated, with ArI^- as the stable product. This picture is suggested by molecular dynamics simulations carried out by Amar[18] on $Br_2^-(CO_2)_n$ clusters. However, recent molecular dynamics simulations by Faeder *et al.*[19,20] suggest a somewhat different mechanism. Their calculations predict that the equilibrium geometry of $I_2^-(Ar)_6$ is an open, highly symmetric structure consisting of a ring of Ar atoms lying in the plane that bisects the I_2^- bond. When the I_2^- is dissociated, the I and I^- fragments separate sufficiently rapidly so that the Ar atoms do not cluster around the I^- fragment. Instead, the departing I^- fragment abstracts one of the solvent atoms, on average, as the cluster breaks up. The shifts in the PE spectrum during the second time regime are qualitatively consistent with this picture, in that as the I^- fragment leaves the cluster, its attractive interactions with the solvent atoms decrease and the electron affinity drops.

We next consider the $I_2^-(Ar)_{20}$ clusters, for which caging is complete.[11] The overall appearance and evolution of these spectra from 300 fs to 1 ps is similar to that seen for $I_2^-(Ar)_6$, in that there are two "I^-" peaks that shift towards higher electron energy

as τ increases. Thus, up to τ=1 ps, the cluster contains I and I⁻ fragments that are essentially independent of one another. The shifting of the peaks towards higher eKE can again be explained as a progressive weakening of the interactions between the I⁻ fragment and the solvent atoms. This is probably due to a combination of evaporation of solvent atoms induced by the recoil energy of the I and I⁻ fragments (~ 0.6 eV), and the rather large excursions that the I⁻ fragment makes within the cluster as the I and I⁻ photoproducts separate on the repulsive $\tilde{A}'\ ^2\Pi_{g,1/2}$ state. Molecular dynamics simulations by Batista and Coker[21] predict the inter-iodine separation increases to 8-10 Å after 1 ps has elapsed, a distance comparable to the original size of the cluster, and it is likely that the strength of the solvent interactions with the I⁻ decreases while this occurs.

The evolution of the $I_2^-(Ar)_{20}$ spectra at later times can be explained as a result of recombination of the I and I⁻ on the two lowest potential energy curves in Fig. 1. The shifting of the two "I⁻" peaks towards lower energy from τ=1 to 35 ps is consistent with recombination and vibrational relaxation on the $\tilde{A}\ ^2\Pi_{g,3/2}$ curve. Recent ab initio calculations predict that R_e = 4.18 Å and D_e=0.11 eV for this state.[22] Fig. 1 shows that at such a large internuclear distance, photodetachment will access the neutral potential energy curves near their asymptotic energies. This will yield two peaks approximately separated by the I atom spin-orbit splitting, but shifted toward lower electron energy compared to bare I⁻ by the well depth (D_e) of the $\tilde{A}\ ^2\Pi_{g,3/2}$ state and the sum of the attractive interactions with the remaining Ar atoms. If we use an approximate binding energy of 73 meV/Ar,[11] the total energy released by the recoiling photofragments and by vibrational relaxation of the I_2^- to the v=0 level of the *ab initio* $\tilde{A}\ ^2\Pi_{g,3/2}$ state is

sufficient to evaporate 9 Ar atoms, so this excited state recombination mechanism is the likely origin of the $I_2^-(Ar)_{<n>=11}$ product seen by Lineberger and co-workers. From these considerations, it is reasonable to attribute the signal at eKE>1.6 eV to recombination on the $\tilde{X}\ ^2\Sigma_u^+$ state followed by vibrational relaxation. This can release enough energy to evaporate all of the Ar atoms, leaving $I_2^-(v=8)$ in the limit of zero photofragment KE.

In order to test these assignments, the spectrum at 200 ps was simulated assuming photodetachment to occur from $I_2^-(Ar)_{11}$ with the I_2^- chromophore in the v=0 level of the $\tilde{A}\ ^2\Pi_{g,3/2}$ state, and from $I_2^-(\tilde{X}\ ^2\Sigma_u^+)$ in a mixture of vibrational levels. The simulations involve calculating the Franck-Condon factors between the anion and neutral vibrational wave functions and scaling the results for different electronic transitions to best match the experimental intensities. For the $\tilde{A}\ ^2\Pi_{g,3/2}$ state, R_e and D_e were taken to be 4.5 Å and 0.16 eV, respectively, with both values differing somewhat from the ab initio values; these differences may reflect in part the influence of the remaining Ar atoms in the cluster. Best results for the $\tilde{X}\ ^2\Sigma_u^+$ state were obtained using a vibrational distribution with $<v>$=5. The results, shown in Fig. 2c, reproduce the experimental spectrum quite well. The $I_2^-(\tilde{A}\ ^2\Pi_{g,3/2}, v=0)$ and $I_2^-(\tilde{X}\ ^2\Sigma_u^+, v=5)$ vibrational wave functions are superimposed on the appropriate potential energy curves in Fig. 1. Note that photodetachment from the inner turning point of the $I_2^-(\tilde{X}\ ^2\Sigma_u^+, v=5)$ wave function is responsible for the signal at eKE > 1.6 eV; the outer turning point contributes to the "I" peak at 1.35 eV along with photodetachment from the $I_2^-\ \tilde{A}\ ^2\Pi_{g,3/2}$ state.

The shifting of the signal at eKE > 1.6 eV toward lower energy from 10-200 ps is attributed to vibrational relaxation of the ground state I_2^- accompanied by evaporative cooling of the Ar atoms, that is a series of reactions of the type $I_2^-(v'')Ar_n \rightarrow I_2^-(v'<v'')Ar_{n-1}+Ar$. As the I_2^- vibrational quantum number decreases, the contribution to the PE spectrum from the inner turning point of the vibrational wavefunction shifts towards lower eKE. Although this is partially compensated by the lowering of the electron affinity as Ar atoms evaporate, our simulations of the PE spectra show a net shift of the signal in this region towards lower eKE as the cluster cools. By comparing these simulations with the experimental spectra, and starting with the I_2^- vibrational distribution at 200 ps, one can estimate the average number of Ar atoms $\langle n \rangle$ and I_2^- vibrational quantum number $\langle v \rangle$ for $\tau<200$ ps. We find that $\langle n \rangle=2$, $\langle v \rangle=17$ at $\tau=35$ ps, and $\langle n \rangle=4.5$, $\langle v \rangle=32$ at $\tau=10$ ps. It therefore appears that 15 or 16 of the original 20 solvent atoms evaporate in the first 10 ps, and that the evaporation rate slows down markedly at later times.

5. Conclusions

Our spectra yield the following picture of the dynamics resulting from photoexcitation of $I_2^-(Ar)_{20}$. As with I_2^- and $I_2^-(Ar)_6$, dissociation of the I_2^- chromophore is complete by 300 fs. Between 300 fs and 1 ps, the interaction between the I and I^- atoms within the cluster is very weak. After 1 ps, the I and I^- atoms recombine on either of the two lower-lying attractive potential energy surfaces. Recombination on the $\tilde{A}\ ^2\Pi_{g,3/2}$ state leads to $I_2^-(Ar)_{\langle n \rangle=11}$ product in which the I_2^- is vibrationally cold, whereas recombination aon the $\tilde{X}\ ^2\Sigma_u^+$ state results in bare I_2^- with $\langle v \rangle=5$. The first

process is complete by 35 ps, whereas the second, involving considerably more energy flow between the I_2^- and the solvent atoms, is over after 200 ps. Although recombination on the $\tilde{A}\ ^2\Pi_{g,3/2}$ state was proposed as a possible mechanism in Lineberger's earlier study[11], our experiments provide conclusive spectroscopic evidence that this occurs.

6. Acknowledgments

This research is supported by the National Science Foundation under Grant No. CHE-9404735. Support from the Defense University Research Instrumentation Program under Grant No. F49620-95-1-0078 is also gratefully acknowledged. The authors thank J. Faeder, P. Maslen, V. Batista, and R. Parson for helpful discussions and for providing access to unpublished results. We are grateful to R. J. Le Roy for a copy of RKR1: A Computer Program for Implementing the First-Order RKR Method for Determining Diatom Potential Energy Curves from Spectroscopic Constants.

7. References

1 A. H. Zewail, J. Phys. Chem. **100,** 12701 (1996).

2 P. L. Houston, J. Phys. Chem. **100,** 12757 (1996).

3 M. H. Head-Gordon, J. Phys. Chem. **100,** 13213 (1996).

4 G. C. Schatz, J. Phys. Chem. **100,** 12839 (1996).

5 A. W. Castleman, Jr. and K. H. Bowen, Jr., J. Phys. Chem. **100,** 12911 (1996).

6 B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Chem. Phys. Lett. **258,** 523 (1996).

7 D. R. Cyr and C. C. Hayden, J. Chem. Phys. **104,** 771 (1996).

8 P. Ludowise, M. Blackwell, and Y. Chen, Chem. Phys. Lett. **258,** 530 (1996).

9 A. Assion, M. Geisler, J. Helbing, V. Seyfried, and T. Baumert, Phys. Rev. A **54,** R4605 (1996).

10 J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, H. K. Buchenau, and W. C. Lineberger, J. Chem. Phys. **99,** 8733 (1993).

11 V. Vorsa, P. J. Campagnola, S. Nandi, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **105,** 2298 (1996).

12 V. Vorsa, S. Nandi, P. J. Campagnola, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **106,** 1402 (1997).

13 Q. L. Liu, J.-K. Wang, and A. H. Zewail, Nature **364,** 427 (1993).

14 M. T. Zanni, T. R. Taylor, B. J. Greenblatt, B. Soep, and D. M. Neumark, J. Chem. Phys. **107,** 7613 (1997).

15 E. C. M. Chen and W. E. Wentworth, J. Phys. Chem. **89,** 4099 (1985).

16 D. R. T. Appadoo, R. J. Leroy, P. F. Bernath, S. Gerstenkorn, P. Luc, J. Verges, J. Sinzelle, J. Chevillard, and Y. Daignaux, J. Chem. Phys. **104,** 903 (1996).

17 I. Yourshaw, Y. Zhao, and D. M. Neumark, J. Chem. Phys. **105,** 351 (1996).

18 L. Perera and F. G. Amar, J. Chem. Phys. **90,** 7354 (1989).

19 J. Faeder and R. Parson, J. Chem. Phys. **108,** 3909 (1998).

20 J. Faeder, N. Delaney, P. E. Maslen, and R. Parson, Chem. Phys. Lett. **270,** 196 (1997).

21 V. S. Batista and D. F. Coker, J. Chem. Phys. **106,** 7102 (1997).

22 P. E. Maslen, J. Faeder, and R. Parson, Chem. Phys. Lett. **263,** 63 (1996).

Chapter 5. Time-resolved Studies of Dynamics in Molecular and Cluster Anions*

Femtosecond photoelectron spectroscopy (FPES) is used to study the time-resolved photodissociation dynamics of $I_2^-(CO_2)_{n=4,16}$ clusters excited at 780 nm. The FPES experiment on $I_2^-(CO_2)_4$ shows that the I^- fragment formed by excitation to the A' $^2\Pi_{1/2,g}$ repulsive state of I_2^- initially pulls away from the cluster, but by 0.2 ps it is drawn back to complex with more of the solvent molecules. In the n=16 cluster, where caging of the I_2^- is known to be complete, FPES probes the recombination dynamics of the I_2^- in considerable detail. Specifically, vibrational relaxation on the I_2^- $X\ ^2\Sigma_u^+$ state and the accompanying evaporation on CO_2 molecules can be followed in real-time. Vibrational relaxation is essentially complete by 10 ps, whereas solvent evaporation is not entirely complete by 200 ps. The spectra also show evidence for short-lived recombination on the I_2^- $A\ ^2\Pi_{3/2,g}$ state. The results are compared to previous experimental results for $I_2^-(Ar)_n$ clusters and recent simulations of cluster dynamics.

1. Introduction

The effect of clustering on the dynamics of elementary chemical processes has been the focus of considerable interest, as it offers a route toward understanding the evolution of chemistry from gas phase to condensed phase environments. Much of the original work in this area focused on neutral van der Waals clusters,[1-3] in which a chromophore such as I_2 was complexed to one or more solvating species, and the resulting effects on the chromophore photophysics were probed using laser-induced

*B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, *Faraday Discuss.*, **108**, 101 (1997).

fluorescence, multiphoton ionization, and other spectroscopic/dynamical probes. More recently, femtosecond time-resolved techniques have been applied to clusters of this type.[4,5] A parallel effort has developed in the study of ionic clusters comprised of solvated charged chromophores.[6] These experiments have an advantage over neutral cluster studies in that there is generally no ambiguity concerning the size of the cluster, an important issue if one is trying to probe size-dependent effects. Clusters of I_2^- with Ar and CO_2 have been of particular recent interest; the photodissociation dynamics of these clusters have been studied in an elegant series of frequency and time-resolved experiments by Lineberger and co-workers.[7-14] We have undertaken studies of these clusters using a new experimental technique, femtosecond photoelectron spectroscopy (FPES), providing a picture of the photodissociation dynamics that in many ways complements the Lineberger experiments. Previously we have reported results on I_2^- and $I_2^-(Ar)_n$ clusters.[15,16] Here new results for $I_2^-(CO_2)_n$ clusters are presented and discussed in light of earlier experiments on $I_2^-(Ar)_n$ clusters.

Two types of experiments have been performed by Lineberger's group on clusters of I_2^- and other dihalides with CO_2 and Ar. In the experiments on I_2^- (see Fig. 1), the anion is photoexcited from the $X\ ^2\Sigma_u^+$ state to a repulsive electronic state, the $A'^2\Pi_{1/2,g}$ state, and the subsequent interactions between the recoiling photofragments and solvent species (S) are probed. In one set of experiments,[7,8,13,22] the product masses from one-photon dissociation were determined. These experiments show that for small numbers of solvent species, only "uncaged" ionic products of the type $I^-(S)_n$ are produced, whereas for larger clusters, stable products of the type $X_2^-(S)_n$ dominate. These "caged" products result from recombination of the photofragments on a lower-lying, bound state of I_2^-, a

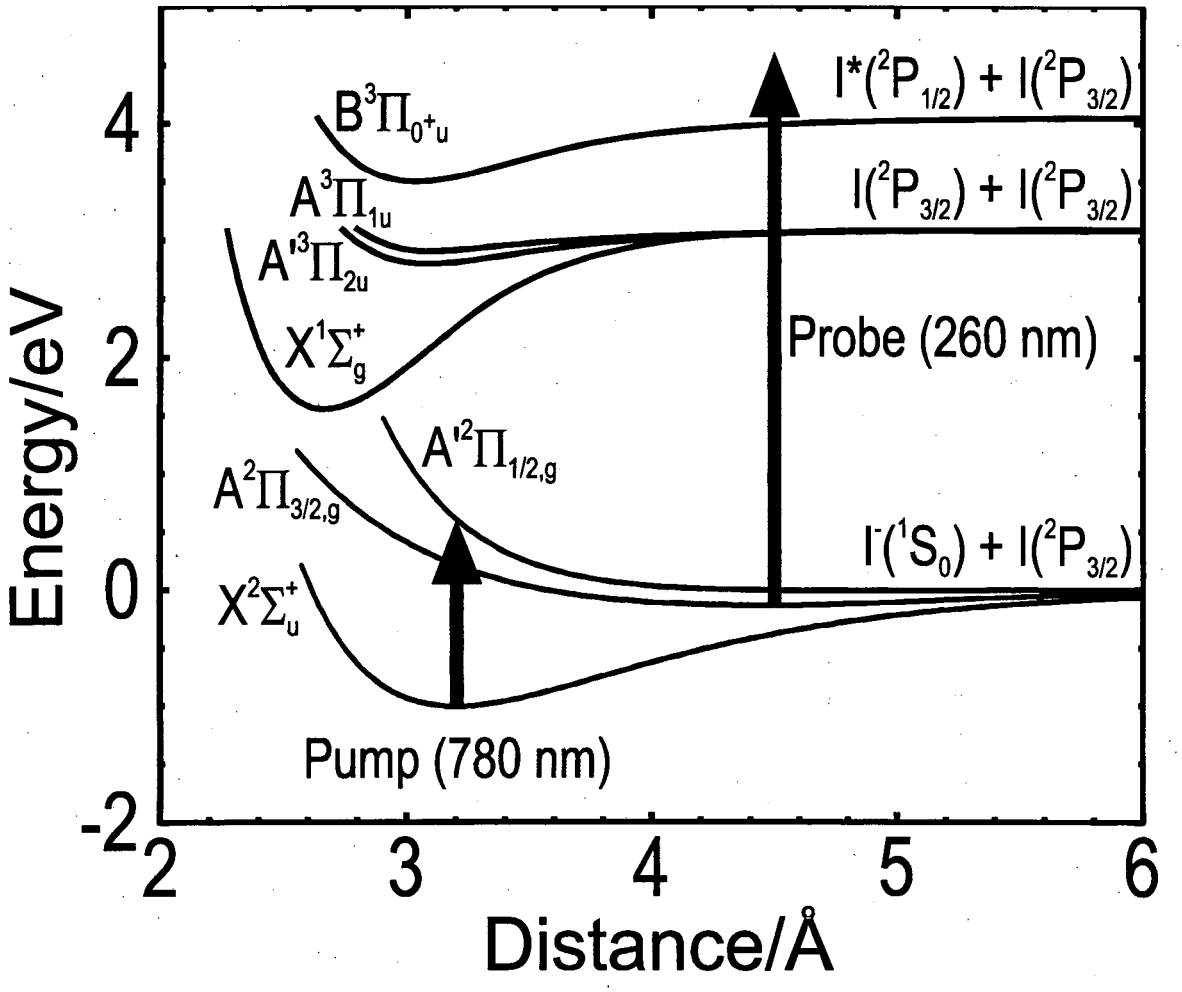


Fig. 1. Potential energy curves for low-lying electronic states of I_2^- and I_2. Correlating atomic states are indicated to the right. The anion $X\,^2\Sigma_u^+$ and $A\,^2\Pi_{3/2,g}$ state parameters are taken from Ref. [16]. The anion $A'\,^2\Pi_{1/2,g}$ state parameters are taken from Ref. [17]. Neutral state parameters are from Refs. [18-21].

process analogous to geminate recombination in solution.[23] The second set of experiments[9-11,14] used a two-photon pump-and-probe scheme with picosecond and, more recently, femtosecond lasers to perform time-resolved studies of the recombination dynamics in clusters of I_2^- sufficiently large so that caging occurs. In these experiments, the I_2^- chromophore is dissociated with the pump pulse, and subsequent absorption of the probe pulse (of the same color) is monitored as a function of delay time. The experiments yield the overall time scale for recombination and vibrational relaxation of the I_2^- on its

ground electronic state: 1.3 ps in the case of $I_2^-(CO_2)_{16}$, versus 130 ps for $I_2^-(Ar)_{20}$.[14] While the recovery of the I_2^- absorption is monotonic for $I_2^-(Ar)_n$ clusters, the results for $I_2^-(CO_2)_{16}$ have been interpreted in terms of "coherent recombination" of the photofragments occurring on the $A\,^2\Pi_{3/2,g}$ excited state of I_2^- (see Fig. 1) ~2 ps after the pump pulse.[9,10,14] In related work, the time-resolved recombination dynamics of I_2^- in solution were studied by Barbara and co-workers via transient absorption in a variety of polar solvents.[24,25] The absorption recovery times lie in the range of a few picoseconds, i.e. a similar time scale to the $I_2^-(CO_2)_n$ clusters.

The finite clusters have been considered in a series of theoretical papers in which molecular dynamics simulations are used to determine the equilibrium geometries of the clusters and track the dynamics subsequent to photodissociation of the dihalide chromophore. The original studies by Perera and Amar[26] focused on the time scales for recombination and solvent evaporation on the ground electronic state of the dihalide. More recent work by Batista and Coker[27] and Parson and co-workers[28-30] has considered the importance and time-scale of non-adiabatic electronic transitions that occur subsequent to photoexcitation. Parson in particular has emphasized the role of "anomalous charge switching" in these clusters, in which the asymmetric charge distribution on the two iodine atoms induced by solvation in the cluster ground state is reversed in the photoexcited state.

The FPES experiments discussed here were undertaken to provide a more complete picture of the dissociation dynamics in $I_2^-(CO_2)_n$ clusters. In these experiments, the I_2^- chromophore is excited to the repulsive $A'\,^2\Pi_{1/2,g}$ state by an ultrafast (~80 fs) pump pulse. The time-evolution of the cluster is monitored by photodetachment with an

ultrafast probe pulse and measurement of the resulting photoelectron spectrum. At each pump-probe delay, the photoelectron spectrum provides a "snapshot" of the cluster dynamics, and is particularly sensitive to the local environment of the excess electron in the cluster. In contrast to the Lineberger's pump-probe experiments, FPES can be used to investigate clusters in which no caging and recombination occurs. When recombination does occur, FPES can reveal the electronic state of the dihalide at each delay time, along with the degree of vibrational excitation and the approximate number of solvent species remaining in the cluster. Results are reported here for two clusters: $I_2^-(CO_2)_4$, for which almost no caging occurs, and $I_2^-(CO_2)_{16}$, in which caging is complete.

2. Experimental

Although the FPES experiment has been described in detail elsewhere,[15] several improvements have been made recently and are discussed below. Figure 2 shows the apparatus. Cluster anions are generated by passing a mixture of 3% CO_2 in Ar over I_2 crystals at a backing pressure of 10-30 psig, expanding the gas mixture into a vacuum chamber through a piezoelectric valve running at a repetition rate of 500 Hz, and crossing the resulting free jet with a 1.5 keV electron beam just downstream of the nozzle. The resulting plasma cools collisionally to produce both positive and negative ions. After passing through a 5 mm diameter skimmer located 11 mm below the valve orifice, the negative ions are injected into a Wiley-McLaren time-of-flight mass spectrometer[31] by applying pulsed extraction and acceleration fields perpendicular to the beam axis. The final beam energy varies between 800 eV and 1.7 keV, depending on the voltages used. A three-plate pulsed mass gate[32] insures that only anions of the desired mass interact with the lasers.

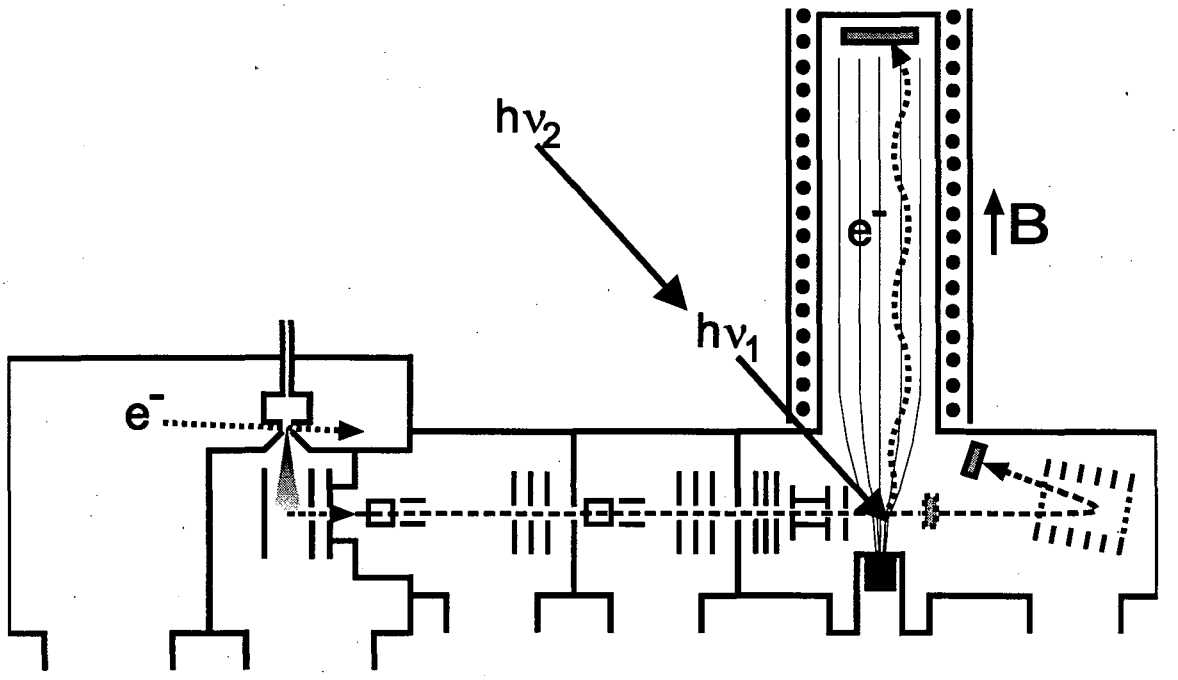


Fig. 2. Schematic of the apparatus. Shown are the ion source, time-of-flight mass spectrometer, "magnetic bottle" photoelectron spectrometer and reflectron photofragment analyzer.

The original source chamber of our apparatus has been divided into two regions to accommodate additional differential pumping. Each region is now pumped by a Varian VHS-10 diffusion pump with 4400 L/s pumping speed; this results in a considerably lower pressure in the region where the ion extraction pulses are applied. On their way to the laser interaction region, the anions pass through two additional differentially pumped regions. The first differential region is pumped by a Varian VHS-6 diffusion pump. The second differential region and laser interaction region are each pumped by Varian V250 turbomolecular pumps. The base pressure in the final region is 1×10^{-9} torr.

Laser pulses cross the ion beam at the focus of a "magnetic bottle" photoelectron spectrometer, which is based on the design of Cheshnovsky *et al.*[32] However, a strong (0.8 tesla) permanent magnet, rather than an electromagnet, is used to produce the inhomogeneous magnetic field. It is located 9.5 mm below the beam axis, outside the

vacuum chamber, and can be easily removed. A 1.3 m long solenoid field (20 gauss) guides the photoelectrons toward a 75 mm diameter dual microchannel plate detector. The arrival time distribution is recorded after each laser shot with a Stanford Research Systems SR430 multichannel scalar. Because of the inherently low resolution (~250 meV) of a spectrometer which collects all of the electrons ejected from a fast-moving ion beam, a pulsed deceleration field is used to slow the ions down just before the interaction region.[33,34] This results in an improvement in the electron energy resolution of up to a factor of four, with further improvements expected shortly.

An in-line microchannel plate detector mounted on a retractable translator arm is used to record time-of-flight mass spectra of the ion beam. We can also measure the photofragment mass spectra resulting from excitation of a particular cluster with the pump pulse alone. To do this, the primary ion detector is retracted, allowing the ions to continue into an off-axis reflectron[7] which separates the daughter and parent ions. These are collected by another microchannel plate detector for photofragment mass analysis. Both types of mass spectra are recorded using a Tektronix TDS744A digitizing oscilloscope at a repetition rate of ~80 Hz.

The pump and probe laser pulses are generated from a commercial femtosecond laser system. A Coherent Innova-90 Ar^+ laser pumps a Clark-MXR NJA-5 Ti:sapphire oscillator. Selected pulses are amplified using a Clark-MXR regenerative amplifier system that includes a pulse stretcher, a Ti:sapphire regenerative amplifier pumped by a Nd:YAG laser running at a repetition rate of 500 Hz, and a pulse compressor. At 780 nm, the pump pulse wavelength, the pulse width and energy are 70 fs FWHM ($sech^2$) and 1 mJ, respectively. About 80% of this beam is directed into a frequency tripling unit

(CSK Optronics 8315A), resulting in a probe pulse at 260 nm with width and energy of 110 fs and 20 μJ, respectively. (The width of the probe pulse is measured by difference frequency cross-correlation using a 300 μm thick KDP crystal). The remainder of the 780 nm pulse passes through a computer-controlled variable delay line, and is then collinearly recombined with the probe pulse prior to entering the vacuum chamber. The polarization of the pump and probe pulses are perpendicular to the ion beam axis. For accurate determination of the temporal overlap of the pulses inside the vacuum chamber, two-color above threshold photodetachment (ATD) of I^- is used.[35]

Because the probe pulse wavelength is sufficient to detach electrons from ground state $I_2^-(Ar)_n$ and $I_2^-(CO_2)_n$ clusters, the photoelectron spectra are not background-free. Background subtraction is accomplished by either alternating 20 s scans between the desired delay and a fixed, negative (–2 ps) delay, or by using an optical chopper (New Focus 3501). The chopper blocks the pump pulse every other laser shot, and the SR430 scalar performs shot-by-shot background subtraction. Background spectra are also collected concurrently at 80 Hz repetition rate with the TDS744A oscilloscope. These are stored and used for longer-time normalization of the spectra. Depletion of the I_2^- ground state[36] causes a bleach of the background-subtracted signal, which is compensated by adding a percentage of the background back to the spectra.

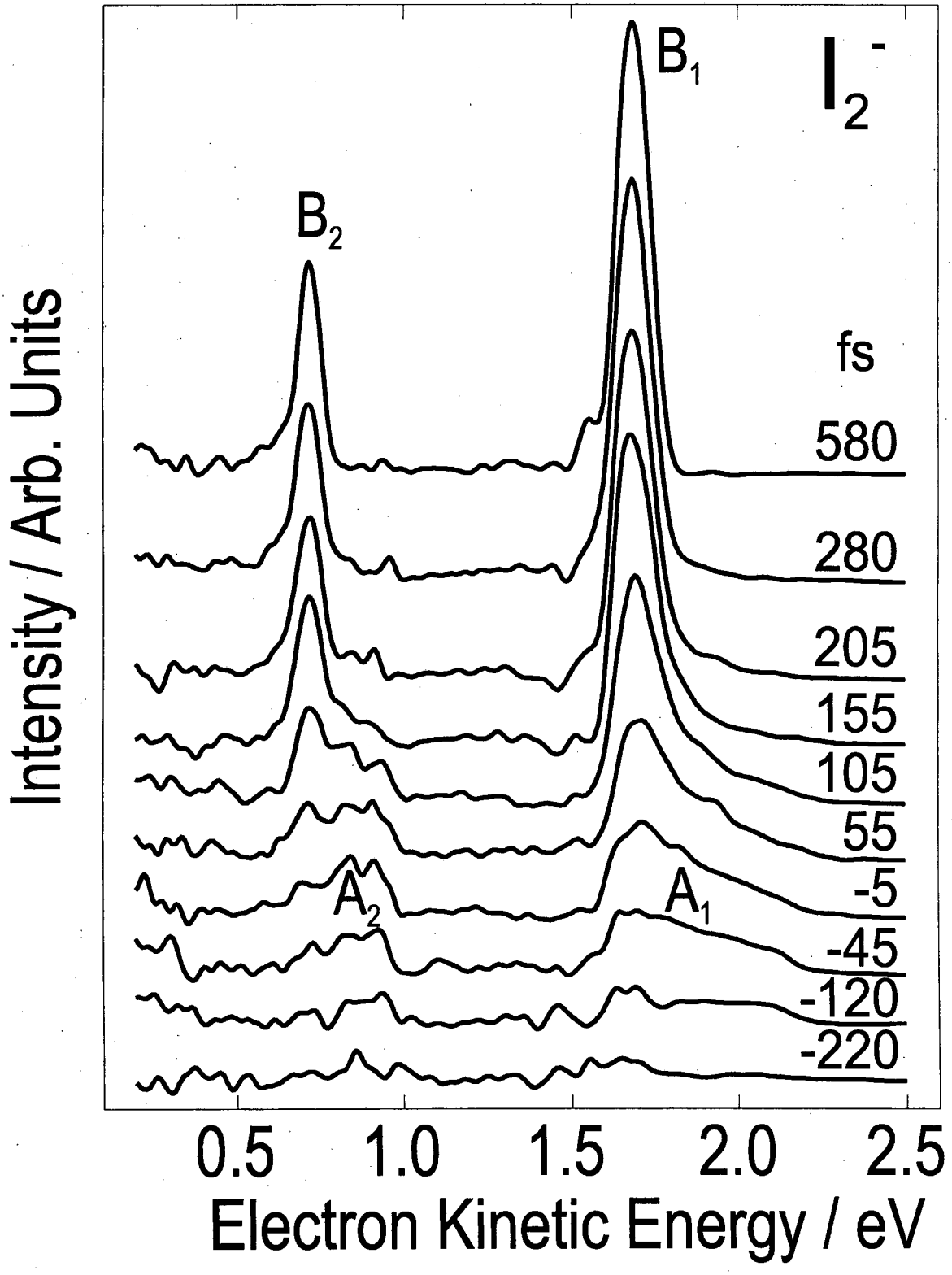


Fig. 3. Femtosecond photoelectron spectra of bare I_2^-, with decelerated ion beam. The pump-probe delay times are indicated to the right of the spectra. Assignments of various features are indicated, and explained in the text.

3. Results

Figure 3 shows FPES spectra of bare I_2^- for several pump-probe delay times. These spectra are taken using pulsed deceleration to slow down the ion beam; consequently the electron energy resolution (~100 meV) is substantially better than in our spectra reported and discussed previously.[15] As the delay time increases, two broad features, A_1 and A_2 shift toward lower electron energy and evolve into two sharp features B_1 and B_2, at electron energies of 1.71 and 0.77 eV, respectively. Peaks B_1 and B_2 represent photodetachment of the I^- photoproduct to the $^2P_{3/2}$ and $^2P_{1/2}$ states of atomic iodine, respectively, whereas the broader features A_1 and A_2 at early times result from photodetachment of the dissociating wavepacket on the $A'^2\Pi_{1/2,g}$ anion state to the close lying $A'^3\Pi_{2u}$ and $A^3\Pi_{1u}$ states (A_1), and the $B^3\Pi_{0^+u}$ state (A_2) of neutral I_2. No evolution of the spectra occurs after 200 fs, indicating that dissociation of the bare ion is complete by this time.

Femtosecond photoelectron spectra for $I_2^-(CO_2)_4$ are shown in Figure 4, also with a decelerated ion beam. At short times, from 0.0 to 0.1 ps, the evolution of the photoelectron signal between 1.4 and 2.0 eV is similar to bare I_2^-, in that a broad feature (A) arises and shifts toward lower electron energy to form a narrower peak (B_1). At lower energy, a second sharp feature (B_2) arises on the same time scale. B_1 and B_2 are separated approximately by the spin-orbit splitting in atomic iodine (0.943 eV), and therefore appear to be analogous to the atomic I^- transitions in Fig. 3, although they are noticeably broader and shifted toward lower electron energy by 0.30 eV. By 0.2 ps, two new features are evident in the spectrum on the low energy side of B_1 and B_2, labeled C_1 and C_2, with each of the new features occurring at 0.14 eV lower electron energy than the

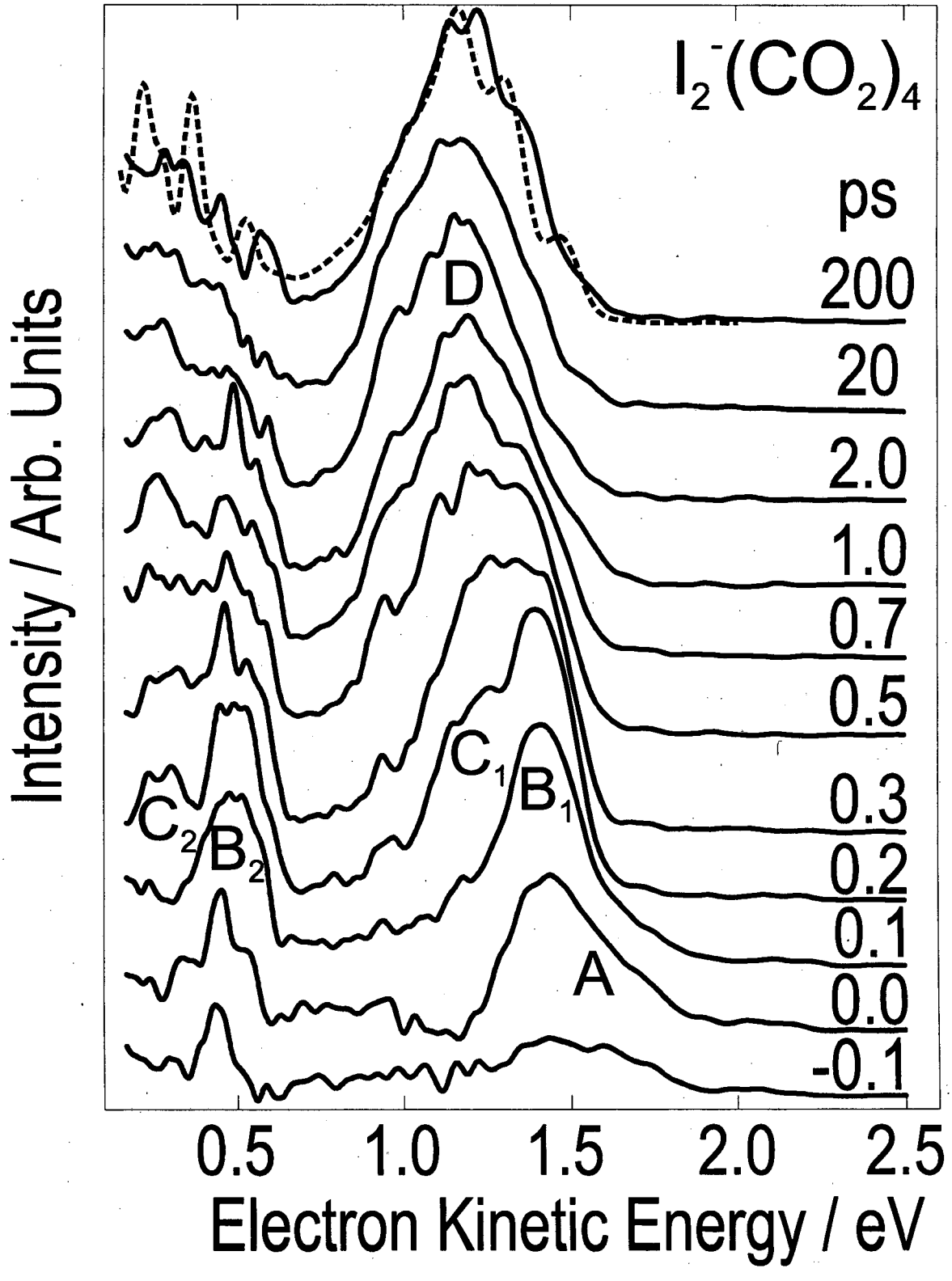


Fig. 4. Femtosecond photoelectron spectra of $I_2^-(CO_2)_4$ (with decelerated ion beam). A simulation (---) of the 200 ps spectrum is shown superimposed on the experimental

spectrum. Labeled features are discussed in the text. Mass distribution used in simulation: n=1, 23%; n=2, 39%; n=3, 30%; n=4, 8%.

main peaks. By 0.5 ps, each doublet has evolved into a single broad peak (D_1 and D_2). D_1 broadens and shifts toward lower electron energy from 0.3-2 ps, followed by a slight shift (0.05 eV) of the entire feature to higher energy between 2 and 200 ps.

Figure 5 shows femtosecond photoelectron spectra for $I_2^-(CO_2)_{16}$. In contrast to the I_2^- and $I_2^-(CO_2)_4$ spectra, no transitions to neutral electronic states correlating to $I(^2P_{1/2})$ are seen; these are too high in energy for the probe pulse because of stabilization energy of the anion from the 16 solvent molecules initially. At 0.0 ps, the spectrum consists of a broad, symmetric feature (A) centered at 0.72 eV, which is analogous to the transient in the FPES of bare I_2^-. As the delay time increases, this feature rapidly disappears, while another broad feature (B) centered at 0.38 eV dominates the spectrum by 0.2-0.4 ps. By 0.7 ps, this feature appears as a shoulder on lower energy feature, labeled C in Fig. 5; this shoulder steadily decreases in intensity and disappears by 4.0 ps. An additional high energy feature (D) is apparent starting at 0.7 ps between 0.5 and 1.7 eV. This feature increases in intensity to 1.6 ps, and from 1.6 to 10 ps shifts gradually toward lower electron energy. During this time, feature C shifts toward higher energy, coalescing with D into a single feature (E) by 10 ps.

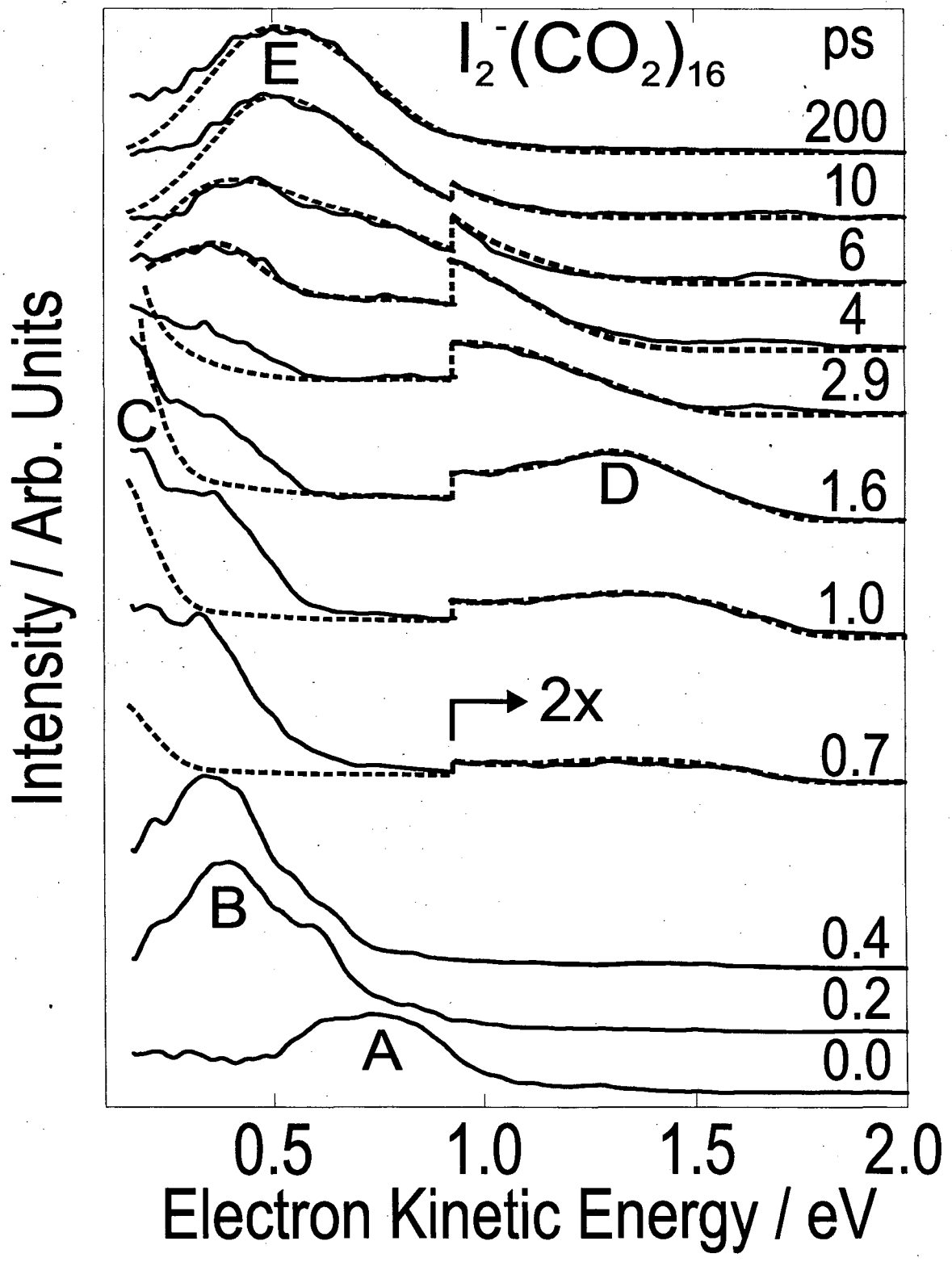


Fig. 5. $I_2^-(CO_2)_{16}$ femtosecond photoelectron spectra. Simulations (---) of spectra at 0.4 ps and later based on parameters in Table I are shown superimposed on experimental spectra. Between 0.4 and 10 ps, the vertical scale is expanded for energies larger than 0.9 eV. Labeled features are are discussed in the text.

4. Discussion

4.1. $I_2^-(CO_2)_4$

It is instructive to compare the FPES results for $I_2^-(CO_2)_4$ with those obtained for $I_2^-(Ar)_6$. [16] Lineberger found that $I^-(CO_2)_2$ and $I^-(Ar)$ are the dominant products from the photodissociation of $I_2^-(CO_2)_4$ at 720 nm and $I_2^-(Ar)_6$ at 790 nm, respectively.[8,13] At 780 nm, we measure essentially the same distribution of products for $I_2^-(CO_2)_4$ using the reflectron mass analyzer to separate the photoproducts from the pump laser alone. In spite of similar asymptotic product distributions for the two anions, with essentially zero caging in both cases, the FPES spectra of $I_2^-(CO_2)_4$ differ significantly from those for $I_2^-(Ar)_6$. The $I_2^-(Ar)_6$ spectra show that the I_2^- bond breaks in approximately 200 fs, just as in bare I_2^-. The resulting "I" features then shift toward higher electron energy from 240 to 1200 fs without otherwise changing in appearance, and do not evolve further after 1200 fs. This is due to a progressive weakening of the interaction between the I^- anion and the Ar solvent atoms as the charged photofragment leaves the cluster.[29]

In the $I_2^-(CO_2)_4$ spectra in Fig. 4, the narrow "I" features, B_1 and B_2, are clearly apparent at 0.1 ps. They are shifted by 0.30 eV toward lower electron energy from bare I^-; this "solvent shift" corresponds to ~1.5 CO_2 molecules.[37] However, the appearance by 0.2 ps of features C_1 and C_2 at lower electron energy indicates that the interaction between the I^- fragment and solvent molecules has *increased* between 0.1 and 0.2 ps, and the subsequent evolution of the doublets into the broad features D_1 and D_2 by 0.5 ps implies that this interaction strengthens further during this time. The spectra thus suggest that the I^- fragment does not monotonically move away from the solvent species, as was the case in $I_2^-(Ar)_6$. Instead, it appears to initially pull away from the cluster (0.1 ps) but

then complexes with the solvent molecules (0.2-0.5 ps). These dynamics are consistent with the considerably deeper well in $I^-...CO_2$ (212 meV)[38] as compared to $I^-...Ar$ (46 meV).[39] Fig. 6 shows a "cartoon" of the dissociation dynamics.

Parson and co-workers have performed molecular dynamics simulations on somewhat larger $I_2^-(CO_2)_n$ clusters which show effects similar to those implied by our spectra.[30] These calculations show that in the *X* state of the cluster, there is an asymmetric charge distribution on the two I atoms; the CO_2 molecules preferentially solvate the I atom with the larger negative charge. This situation is reversed upon excitation to the *A'* state, an effect referred to as "anomalous charge switching".[29] Consequently, once dissociation begins, the I^- fragment is relatively unencumbered by solvent molecules. Although the interiodine distance rapidly increases, the attractive force between the I^- and the CO_2 molecules surrounding the I atom fragment is sufficient to prevent or at least slow down dissociation on the *A'* state, and this attractive force results in the solvent atoms being drawn toward the I^-. The resulting more symmetric distribution of solvent molecules induces non-adiabatic transitions to the lower-lying *A* or *X* state. This is accompanied by rapid, asymmetric solvation of the I^-, leaving the neutral I fragment with its much weaker solvent interaction free to leave the cluster. These calculations therefore suggest that the rapid complexation of the I^- fragment and dissociation of the cluster as evidenced by the evolution of the sharp features B_1 and B_2 into the broader features D_1 and D_2 is associated at least in part with a non-adiabatic transition to one of the two lower-lying electronic states of the cluster.

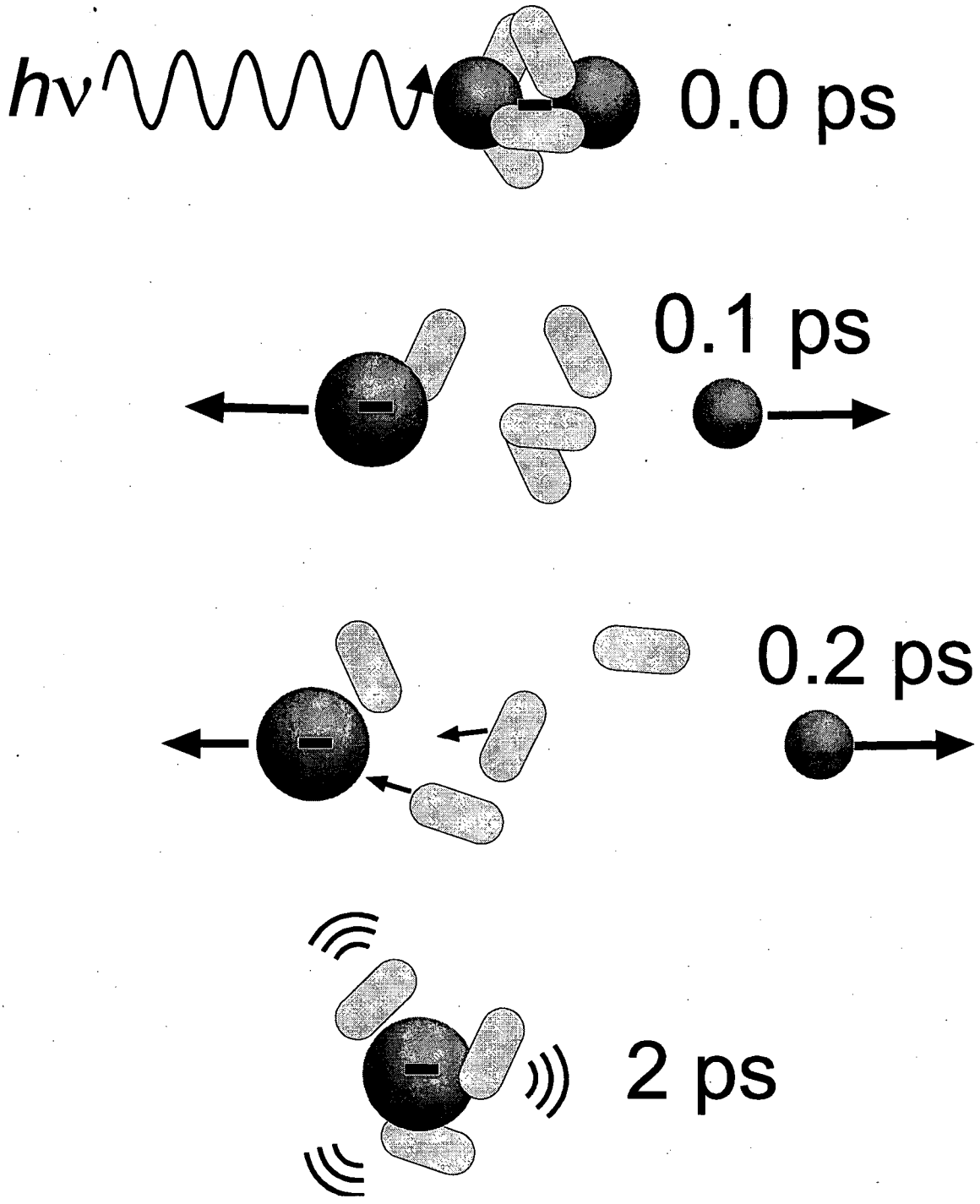


Fig. 6. "Cartoon" of dissociation dynamics in the $I_2^-(CO_2)_4$ cluster. Dark spheres indicate iodine atoms, and light elongated structures denote CO_2 molecules. The symbols ((( ))) indicate vibrational excitation.

Little change in the spectra occurs after 2 ps, so these photoelectron spectra are attributed to $I^-(CO_2)_n$ clusters. In this time regime, the number of CO_2 molecules solvated to the I^- fragment can be estimated by fitting the spectra to a distribution of $I^-(CO_2)_n$ photoelectron spectra; these spectra have been measured previously[37] and show that for $n \leq 9$, each CO_2 molecule increases the electron binding energy by ~150 meV. The results of the best fit at 200 ps is shown superimposed on the experimental spectrum Fig. 4; the assumed distribution is given in the figure caption. Note that the n=2 and n=3 clusters constitute the bulk of the products at 200 ps, with n=2 being slightly dominant. This disagrees with the experimental mass distribution, in which $I^-(CO_2)_2$ and $I^-(CO_2)_3$ comprise 75% and 7% of the products, respectively.[40] This discrepancy may indicate that the time required to evaporate the last CO_2 molecule is longer than the time window of the experiments (200 ps), in contrast to the $I_2^-(Ar)_6$ results in which photoelectron spectra corresponding to the asymptotic ArI^- product was evident by 1.2 ps.[16] This explanation could be tested by measuring spectra at much larger (~ns) delay times, which is feasible with a slight modification to the apparatus. We note that the $I^-(CO_2)_n$ spectra used to fit the spectrum in Fig. 4 were taken for cold anions; the imperfect fit at 200 ps may be an indication that this spectrum is from vibrationally excited $I^-(CO_2)_n$, a necessary condition for further evaporation.

4.2. $I_2^-(CO_2)_{16}$

Previous work on $I_2^-(CO_2)_{16}$ photodissociation at 720 nm and 790 nm by Lineberger and co-workers[8,9,40] show 100% caging of the I_2^- product, with 7 (720 nm) or 6.5 (790 nm) CO_2 molecules lost, on average, via evaporative cooling as the I_2^- recombines and vibrationally relaxes. Time-resolved experiments[9,10,14] show that

relaxation of the I_2^- is complete on a time scale of several picoseconds, with the exact value depending on the photodissociation wavelength. Similar experiments on $I_2^-(Ar)_{20}$ also show 100% caging, but the product mass distribution is bimodal, split approximately evenly between bare I_2^- and $I_2^-(Ar)_{<n>=11}$.[13,40] The I_2^- channel is attributed to recombination on the *X* state of I_2^-, and the FPES study of $I_2^-(Ar)_{20}$ shows that the other channel is due to recombination on the I_2^- *A* state; this state apparently survives for at least several microseconds, the time scale of the Lineberger experiments. The FPES experiments on $I_2^-(Ar)_{20}$ also show that the time scales for vibrational energy relaxation on the *A* and *X* states of I_2^- are 35 and 200 ps, respectively. The role of the *A* state in the dynamics of $I_2^-(CO_2)_{16}$ clusters following photoexcitation appears to be quite different. From the product mass distributions, it is clear that there is no asymptotic trapping on the *A* state. On the other hand, the time-resolved measurements by Lineberger show evidence for "coherent recombination" on the *A* state at pump-probe delays around 2 ps.

With this background, we now consider the interpretation of the $I_2^-(CO_2)_{16}$ spectra in Fig. 5. There are several trends in these spectra to be understood: (1) the evolution and eventual disappearance of feature B from 0.2 to 4 ps, (2), the appearance of features C and D starting at 0.7 ps, and (3) the eventual coalescence of these two features by 10 ps. The second two trends are similar to effects seen in the FPES of $I_2^-(Ar)_{20}$ and are attributed to vibrational relaxation of the I_2^- chromophore on the *X* state potential energy curve. As shown in Figure 7, photodetachment from a highly vibrationally excited anion state results in well-separated high and low energy features in the photoelectron spectrum corresponding to transitions from the inner and outer turning points, respectively, of the vibrational wavefunction on the I_2^- $X\ ^2\Sigma_u^+$ state to the $X\ ^1\Sigma_g^+$ state of neutral I_2. As the I_2^-

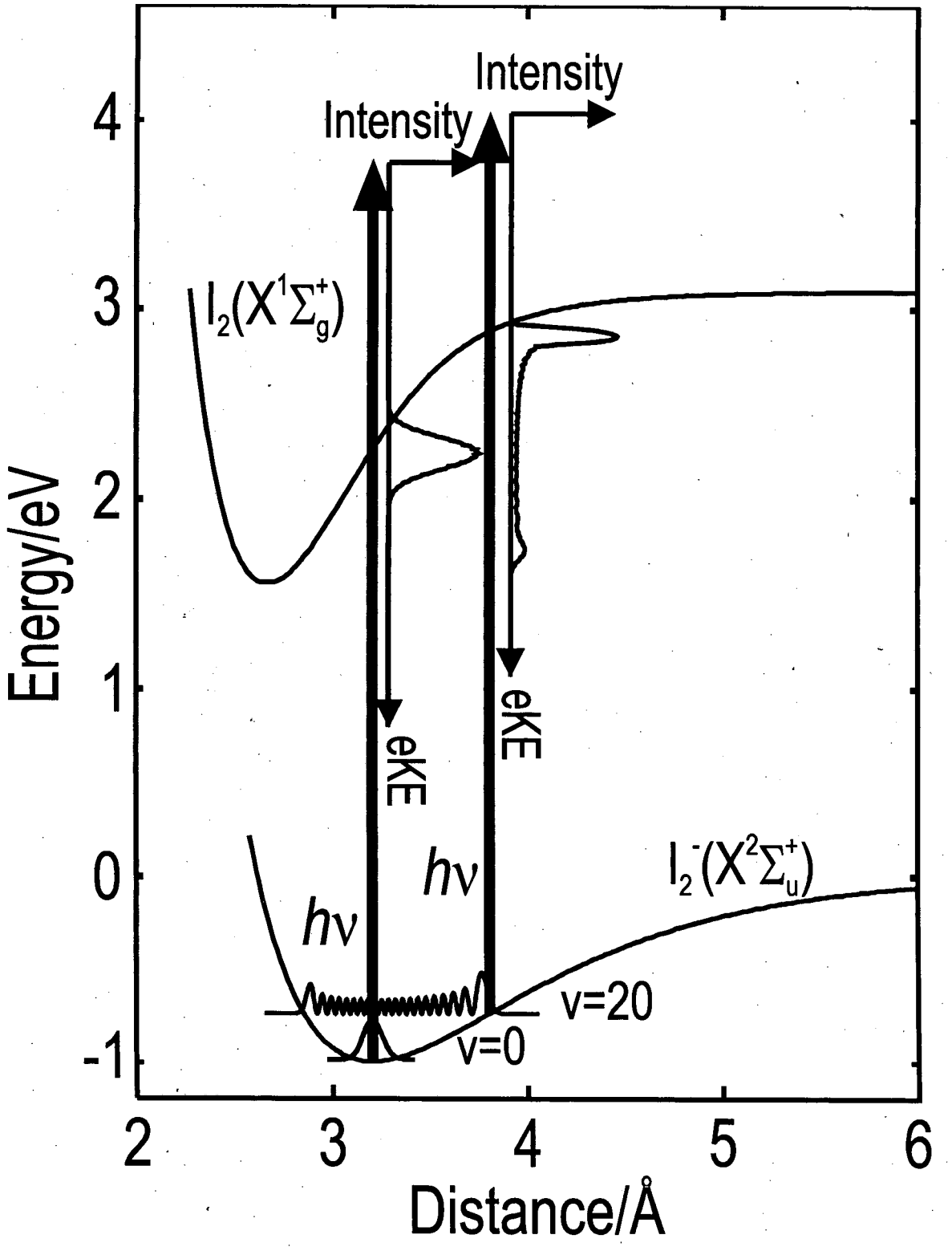


Fig. 7. Simulated I_2^- $X\,^2\Sigma_u^+$ state v=0 and v=20 vibrational wavefunctions, and photoelectron spectra.

vibrationally relaxes, the inner and outer turning points coalesce, as will the two corresponding features in the photoelectron spectrum. Hence, the first appearance of the high energy feature D indicates that recombination on the *X* state has occurred by 0.7 ps, resulting in highly excited I_2^-. The subsequent coalescence of features C and D by 10 ps represents the time scale over which vibrational relaxation is complete. We note that a full coalescence of the analogous features in the $I_2^-(Ar)_{20}$ FPES does not occur, because all of the Ar atoms evaporate before the I_2^- relaxes to its vibrational ground state. In $I_2^-(CO_2)_{16}$, the evaporation of each CO_2 molecule removes considerably more energy from the cluster (~240 vs. 73 meV),[13,40] so I_2^- can easily relax to its ground vibrational state without evaporation of all the solvent molecules.

This process of vibrational relaxation and solvent evaporation can be treated more quantitatively by simulating the FPES at various delay times in order to determine the average level of vibrational excitation and the number of solvent molecules remaining on the cluster as a function of time. To do this, one needs to know how much each CO_2 molecule increases the electron binding energy of the I_2^-. We have measured photoelectron spectra of several $I_2^-(CO_2)_n$ clusters using the probe laser alone, and find an average increase of 80 meV per CO_2 molecule (significantly less than the 140 meV shift for $I^-(CO_2)_n$). Assuming this to be independent of the I_2^- vibrational state, the simulations in Fig. 5 can be generated using a range of vibrational levels and cluster sizes, the average values of which are given in Table I. Thus, for example, at 1.6 ps, the simulations assume a broad vibrational level distribution ($16 \leq v \leq 55$, $\langle v \rangle = 32$) and 13-14 CO_2 molecules solvating the cluster, moving to a much colder distribution ($0 \leq v \leq 17$,

<v>=3) and 11-12 CO_2 molecules by 10 ps. The fit is quite good, except at energies ≤0.4 eV in the spectra between 0.7 and 2.9 ps; this is discussed below.

Time / ps	<v>	$<E_{vib}>$ / eV	<n>
0.7	40.5	0.482	14.5
1.0	40.5	0.482	14.5
1.6	32.1	0.396	13.5
2.9	17.5	0.231	13.5
4	7.3	0.104	11.7
6	4.8	0.071	11.5
10	3.1	0.049	11.5
200	3.1	0.049	11.5

Table I. Average values of parameters used to fit the $I_2^-(CO_2)_{16}$ FPES spectra between 0.7 and 200 ps. <v> = average vibrational level, $<E_{vib}>$ = average vibrational energy, <n> = average number of CO_2 molecules.

The simulations indicate 4-5 CO_2 molecules have evaporated by 200 ps, and that the I_2^- chromophore is largely vibrationally relaxed, with <v>=3. This means nearly all the available energy from relaxation on the X state has been transferred to the various solvent vibrational and librational modes. However, comparison with the photofragmentation study by Vorsa,[40] in which the dominant product fragments are $I_2^-(CO_2)_{9,10}$, indicates that solvent evaporation is not complete by 200 ps. Thus, at 200 ps, the remaining excess energy is distributed among the solvent modes, and the time scale for further solvent evaporation is likely to be described statistically. The incomplete evaporation by 200 ps is consistent with recent simulations by Parson and co-workers, who predict minimum time scales of several hundred ps for complete evaporation.[41]

We next consider the interpretation of feature B. This feature is a distinct peak at 0.2 and 0.4 ps, but from 0.7 to 2.9 ps it appears more as a shoulder in the spectra around 0.4 eV. At 0.2 ps, it is reasonable to assign feature B to newly formed $I^-(CO_2)_n$ within the

cluster; the shift from bare I^- is equivalent to solvation by 8-9 CO_2 molecules. This number does not reflect the number of CO_2 molecules in the cluster, only the average number close enough to the I^- to interact with it. There are two possible interpretations to the subsequent evolution of this feature. One can consider this evolution as a steady decrease of intensity of feature B from 0.4 to 2.9 ps and attribute this decrease to depletion of solvated I^- *via* recombination on the *X* state to form vibrationally excited I_2^-. Alternatively, the change in appearance of feature B from a distinct peak at 0.4 ps to a shoulder at 0.7 ps can be interpreted as recombination on the *A* state, with the disappearance of the shoulder between 0.7 and 4 ps due to leakage out of the *A* state and onto the *X* state. According to this mechanism, which is depicted in the "cartoon" in Figure 8, recombination on both the *X* and *A* state occurs starting around 0.7 ps, but no population remains on the *A* state by 4.0 ps.

The second mechanism is more in line Lineberger's experiments and Parson's simulations,[30] both of which suggest that recombination on the *A* state plays a role in the overall dynamics. In contrast to the photodissociation of $I_2^-(Ar)_{20}$, the stronger interactions with the CO_2 solvent molecules are likely to shorten the lifetime of this excited state significantly, consistent with disappearance of the shoulder by 4 ps. It would also be somewhat surprising for the solvated I^- to persist for several picoseconds, given that recombination in $I_2^-(Ar)_{20}$ occurs in 1 ps, and all other processes common to both clusters occur more rapidly in clusters with CO_2. We therefore favor the mechanism involving some short-lived recombination on the *A* state. However, to really distinguish the two mechanisms it is necessary to have a better understanding of the *A* state and how I_2^- molecules in that state interact with CO_2 solvent molecules.

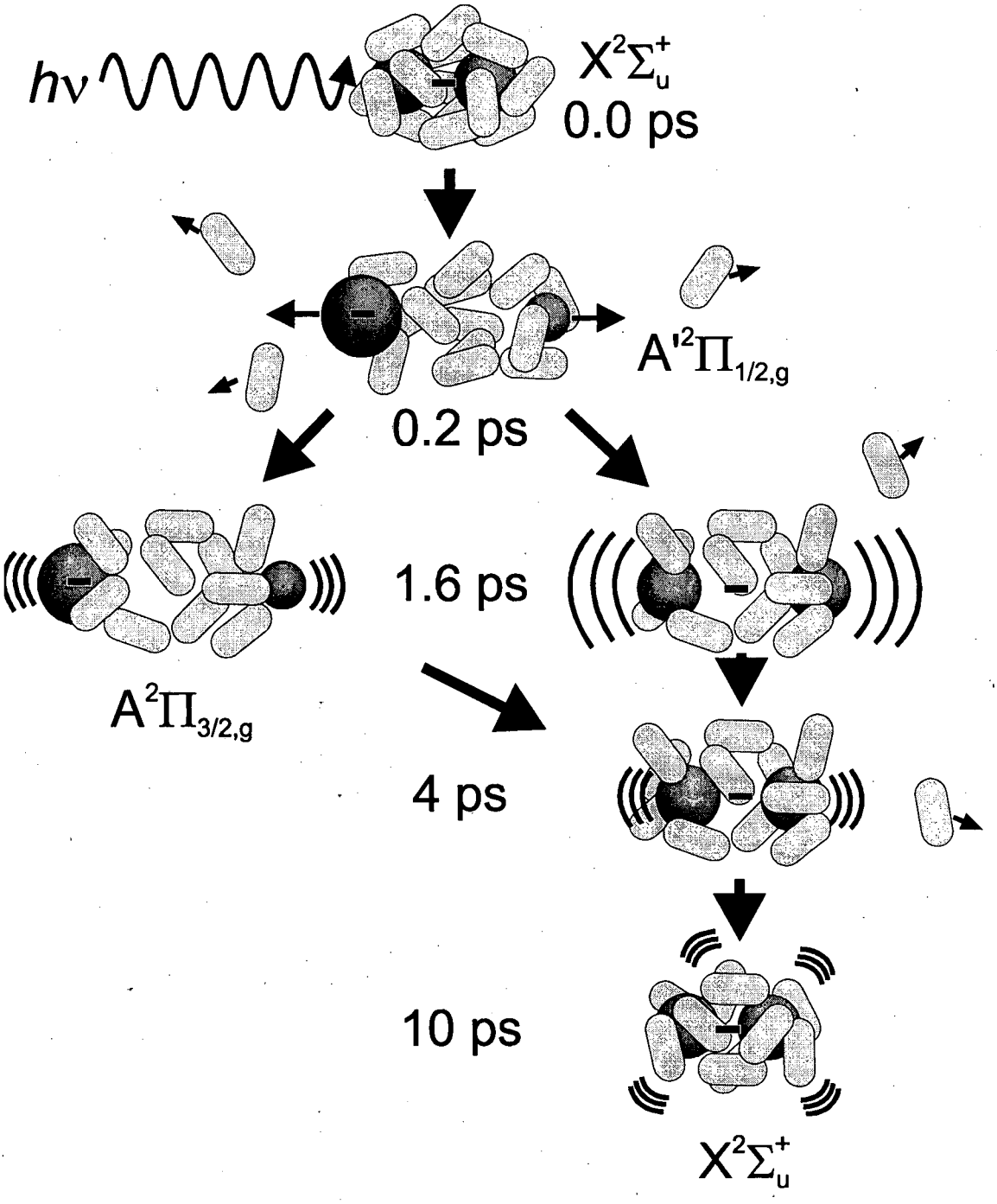


Fig. 8. "Cartoon" of $I_2^-(CO_2)_{16}$ cluster evaporation and recombination dynamics. Symbols are identical to those in Fig. 6.

5. Conclusions

Time-resolved photodissociation studies of $I_2^-(CO_2)_{n=4,16}$ clusters have been performed using femtosecond photoelectron spectroscopy (FPES). The $I_2^-(CO_2)_4$ spectra show that the I^- photofragment initially moves away from the cluster, but the attractive interaction between the I^- and CO_2 molecules is sufficiently strong so that the I^- is prevented from escaping. Instead, from 0.2 to 0.5 ps, it is drawn toward the solvent molecules and complexes with several of them. This differs from the scenario for $I_2^-(Ar)_6$ photodissociation, in which the attraction between the I^- and Ar atoms is sufficiently weak so the anion solvent interaction decreases monotonically subsequent to photodissociation of the I_2^- chromophore. The FPES of $I_2^-(CO_2)_4$ for times greater than 0.7 ps appear to be from a distribution of $I^-(CO_2)_n$ clusters, with the n=2 and 3 clusters present in approximately equal amounts as long as 200 ps after the dissociation pulse. Comparison with photofragment ion mass spectra taken several microseconds after dissociation indicates that solvent evaporation is incomplete at 200 ps.

In $I_2^-(CO_2)_{16}$, the FPES experiment allows us to follow a complex series of events that occurs subsequent to photodissociation of the I_2^- chromophore. Dissociation results in a partially solvated I^- chromophore which can be distinctly observed out to 0.4 ps. We interpret the spectra at longer times to indicate that recombination occurs on both the *A* and *X* states of I_2^-. Recombination on the *A* state is short-lived, and by 4 ps only the *X* state is populated. Starting at 0.7 ps, we can monitor the process of vibrational relaxation on the *X* state and the accompanying evaporation of solvent molecules. We find vibrational relaxation to be largely complete by 10 ps, but solvent evaporation is not

complete even by 200 ps. The role of the *A* state is the most uncertain component of our interpretation and requires further experimental theoretical investigation.

6. Acknowledgments

This work is supported by the National Science Foundation under Grant No. CHE-9710243 and the Defense University Research Instrumentation Program under Grant No. F49620-95-1-0078.

7. References

1 R. E. Smalley, L. Wharton, and D. H. Levy, J. Chem. Phys. **68,** 671 (1978).

2 R. J. Le Roy and J. S. Carley, Advances in Chemical Physics **42,** 353 (1980).

3 N. Halberstadt and K. C. Janda, *Dynamics of Polyatomic van der Waals Clusters* (Plenum, New York, 1990).

4 J. J. Breen, D. M. Willberg, M. Gutmann, and A. H. Zewail, J. Chem. Phys. **93,** 9180 (1990).

5 Q. L. Liu, J.-K. Wang, and A. H. Zewail, Nature **364,** 427 (1993).

6 A. W. Castleman, Jr. and K. H. Bowen, Jr., J. Phys. Chem. **100,** 12911 (1996).

7 M. L. Alexander, N. E. Levinger, M. A. Johnson, D. Ray, and W. C. Lineberger, J. Chem. Phys. **88,** 6200 (1988).

8 J. M. Papanikolas, J. R. Gord, N. E. Levinger, D. Ray, V. Vorsa, and W. C. Lineberger, J. Phys. Chem. **95,** 8028 (1991).

9 J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, J. R. Gord, and W. C. Lineberger, J. Chem. Phys. **97,** 7002 (1992).

[10] J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, H. K. Buchenau, and W. C. Lineberger, J. Chem. Phys. **99,** 8733 (1993).

[11] D. Ray, N. E. Levinger, J. M. Papanikolas, and W. C. Lineberger, J. Chem. Phys. **91,** 6533 (1989).

[12] A. Sanov, S. Nandi, and W. C. Lineberger, J. Chem. Phys. **108,** 5155 (1998).

[13] V. Vorsa, P. J. Campagnola, S. Nandi, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **105,** 2298 (1996).

[14] V. Vorsa, S. Nandi, P. J. Campagnola, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **106,** 1402 (1997).

[15] B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Chem. Phys. Lett. **258,** 523 (1996).

[16] B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Science **276,** 1675 (1997).

[17] E. C. M. Chen and W. E. Wentworth, J. Phys. Chem. **89,** 4099 (1985).

[18] D. R. T. Appadoo, R. J. Leroy, P. F. Bernath, S. Gerstenkorn, P. Luc, J. Verges, J. Sinzelle, J. Chevillard, and Y. Daignaux, J. Chem. Phys. **104,** 903 (1996).

[19] X. N. Zheng, S. L. Fei, M. C. Heaven, and J. Tellinghuisen, J. Chem. Phys. **96,** 4877 (1992).

[20] J. W. Tromp and R. J. Le Roy, J. Mol. Spectrosc. **109,** 352 (1985).

[21] F. Martin, R. Bacis, S. Churassy, and J. Verges, J. Mol. Spectrosc. **116,** 71 (1986).

[22] M. E. Nadal, P. D. Kleiber, and W. C. Lineberger, J. Chem. Phys. **105,** 504 (1996).

[23] D. E. Smith and C. B. Harris, J. Chem. Phys. **87,** 2709 (1987).

[24] A. E. Johnson, N. E. Levinger, and P. F. Barbara, J. Phys. Chem. **96,** 7841 (1992).

25 P. K. Walhout, J. C. Alfano, K. A. M. Thakur, and P. F. Barbara, J. Phys. Chem. **99,** 7568 (1995).

26 L. Perera and F. G. Amar, J. Chem. Phys. **90,** 7354 (1989).

27 V. S. Batista and D. F. Coker, J. Chem. Phys. **106,** 7102 (1997).

28 J. M. Papanikolas, P. E. Maslen, and R. Parson, J. Chem. Phys. **102,** 2452 (1995).

29 J. Faeder, N. Delaney, P. E. Maslen, and R. Parson, Chem. Phys. Lett. **270,** 196 (1997).

30 N. Delaney, J. Faeder, P. E. Maslen, and R. Parson, J. Phys. Chem. A **101** (1997).

31 W. C. Wiley and I. H. McLaren, Rev. Sci. Instrum. **26,** 1150 (1955).

32 O. Cheshnovsky, S. H. Yang, C. L. Pettiette, M. J. Craycraft, and R. E. Smalley, Rev. Sci. Instrum. **58,** 2131 (1987).

33 L.-S. Wang, H.-S. Cheng, and J. Fan, J. Chem. Phys. **102,** 9480 (1995).

34 H. Handschuh, G. Gantefor, and W. Eberhardt, Rev. Sci. Instrum. **66,** 3838 (1995).

35 M. D. Davidson, B. Broers, H. G. Muller, and H. B. van Linden van den Heuvell, J. Phys. B **25,** 3093 (1992).

36 M. T. Zanni, T. R. Taylor, B. J. Greenblatt, B. Soep, and D. M. Neumark, J. Chem. Phys. **107,** 7613 (1997).

37 D. W. Arnold, S. E. Bradforth, E. H. Kim, and D. M. Neumark, J. Chem. Phys. **102,** 3510 (1995).

38 Y. Zhao, C. C. Arnold, and D. M. Neumark, J. Chem. Phys. Faraday Trans. **89,** 1449 (1993).

39 I. Yourshaw, Y. Zhao, and D. M. Neumark, J. Chem. Phys. **105,** 351 (1996).

40 V. Vorsa, Ph.D. Thesis, University of Colorado, Boulder (1996).

41 N. Delaney, J. Faeder, and R. Parson, private communication .

Chapter 6. Femtosecond photoelectron spectroscopy of $I_2^-(Ar)_n$ photodissociation dynamics (n = 6, 9, 12, 16, 20)*

Femtosecond photoelectron spectroscopy has been used to study the photodissociation of I_2^- embedded in size-selected $I_2^-(Ar)_n$ cluster (n = 6-20). This size range spans the uncaged and fully caged product limits for this reaction. The number of Ar atoms around the nascent I^- product decreases in the first ~1.0 ps, due to separation of I^- from the cluster. At longer time delays, the number increases again in $I_2^-(Ar)_{n \geq 9}$, due to an electronic transition from the $\tilde{A}'$ state to the $\tilde{X}$ and/or $\tilde{A}$ states, followed by solvent rearrangement. In $I_2^-(Ar)_{n \geq 12}$, recombination of I_2^- also occurs, along with vibrational relaxation and evaporation of Ar atoms. Simulations of the photoelectron spectra at different time delays were generated in order to characterize the dynamics in detail. An increasing rate of I_2^- recombination is observed as cluster size increases from n = 12 to 20; however, vibrational relaxation is minimal in clusters smaller than n = 20, due to insufficient energy dissipation by Ar evaporation. In $I_2^-(Ar)_{20}$, energy transfer from I_2^- to Ar atoms through vibrational relaxation is slightly faster than energy loss from the cluster through Ar evaporation, indicating the temporary storage of energy within Ar cluster modes. Results are compared to previous experimental studies of $I_2^-(Ar)_n$ photodissociation, as well as theoretical models.

1. Introduction

The femtosecond photodissociation dynamics of I_2^- in small, mass-selected clusters provides an unprecedented opportunity to study real-time energy transfer

* B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, J. Chem. Phys., in preparation for submission

between solute and solvent in the gas phase. Beginning with the pioneering efforts of the Lineberger group studying $Br_2^-(CO_2)_n$ and $I_2^-(CO_2)_n$ photofragments,[1,2] work has expanded to include photofragmentation studies in related clusters[3-5] including $I_2^-(Ar)_n$,[6,7] femtosecond pump-probe experiments,[2,4,8-13] and several theoretical models.[14-27] Photodissociation of I_2^- in solution has also been performed on the femtosecond time scale,[28-32] allowing for comparison between molecular cluster and bulk environments. The general picture which has emerged is that "caging" of photodissociated X_2^- (I_2^-, Br_2^-, ICl^-) can occur in clusters with only a few solvent molecules, producing recombined (X_2^--based) products. As the cluster size increases, so does the caging fraction, reaching unity in the case of $I_2^-(Ar)_n$ at $n = 17$, somewhat less than one full solvent shell. Rates of recombination also increase with cluster size. Changing the type of solvent reveals a strong dependence on solvent-solute binding energy in the recombination rate ($Ar < CO_2 \approx OCS$). Mechanistic changes are also apparent in different solvent environments.

$I_2^-(Ar)_n$ clusters represent an almost ideal weakly interacting system, with an I_2^--Ar well depth of only 53 meV,[33] much smaller than the I_2^- $\tilde{X}$ state well depth (1.014 eV).[34] Despite the small per atom interaction, the collective effect of many solvent atoms has a strong influence on the photodissociation dynamics. A previous study of this system using femtosecond photoelectron spectroscopy (FPES) investigated the uncaged and fully-caged cluster limits,[12] providing information such as the interaction time of the solvent with dissociating I^-, and time scales for electronic transitions and subsequent vibrational relaxation of I_2^-. This work examines these previous clusters, plus three additional intermediate-sized clusters, one of which $[I_2^-(Ar)_{12}]$ yields both caged and uncaged products, allowing the evolution of the energy transfer dynamics to be followed

across cluster size. A companion paper details results for $I_2^-(CO_2)_n$ clusters over a similar range of sizes.[35]

FPES[36] is a time-resolved, pump-probe scheme in which a cold, mass-selected anion is promoted to an excited electronic state by a femtosecond pump pulse. The resulting wavepacket will evolve on this excited potential surface, generally leading to dissociation or nonadiabatic transitions to other electronic states with possible recombination of fragments. A second, delayed femtosecond probe pulse detaches an electron from the anion, producing a photoelectron spectrum. Since the electron kinetic energy depends on the difference between anion and neutral potential energies, identification of electronic and vibrational states of the anion is possible when the neutral potential energy surfaces are well-characterized. The strength of the technique lies in the ability to observe wavepacket dynamics on multiple electronic states at all time delays, without changing the probe wavelength. This was not possible with the techniques employed by Lineberger and coworkers,[7,11] who only observed asymptotic products, or the time-resolved appearance of vibrationally relaxed I_2^-. These and other previous experimental and theoretical studies will be described briefly below.

Photofragmentation mass spectra of photodissociated $I_2^-(Ar)_n$ clusters were measured by Vorsa *et al.*,[6,7] in which a cold, mass-selected cluster was excited to the $\tilde{A}'$ state of I_2^- (Fig. 1) with a pulsed laser at 790 nm, and the masses of photofragments analyzed using a reflectron. They observed only $I^-(Ar)_n$ fragments in smaller clusters, slowly being replaced by $I_2^-(Ar)_n$ fragments starting at a parent cluster size of $n = 10$, with the $I^-(Ar)_n$ channel vanishing by $n = 17$. The numbers of Ar atoms present in both $I^-(Ar)_n$ and $I_2^-(Ar)_n$ fragments were smaller than that of the parent cluster, with more

atoms lost in the I_2^- fragments. This observation was consistent with the expectation that the available energy in the cluster is dissipated through Ar evaporation, so that the larger energy liberated by the recombination of I_2^- resulted in a smaller number of remaining Ar atoms. Interestingly, two distinct $I_2^-(Ar)_n$ fragment size groupings or "channels" were observed, hypothesized to correspond to I_2^- $\tilde{X}$ and $\tilde{A}$ state products. Two $I^-(Ar)_n$ fragment groupings were also observed in larger ($n \geq 11$) clusters, for which no explanation was given, but Faeder *et al.*[20] later attributed the high-mass channel to dissociation on the $\tilde{X}$ or $\tilde{A}$ state, rather than the $\tilde{A}'$ state.

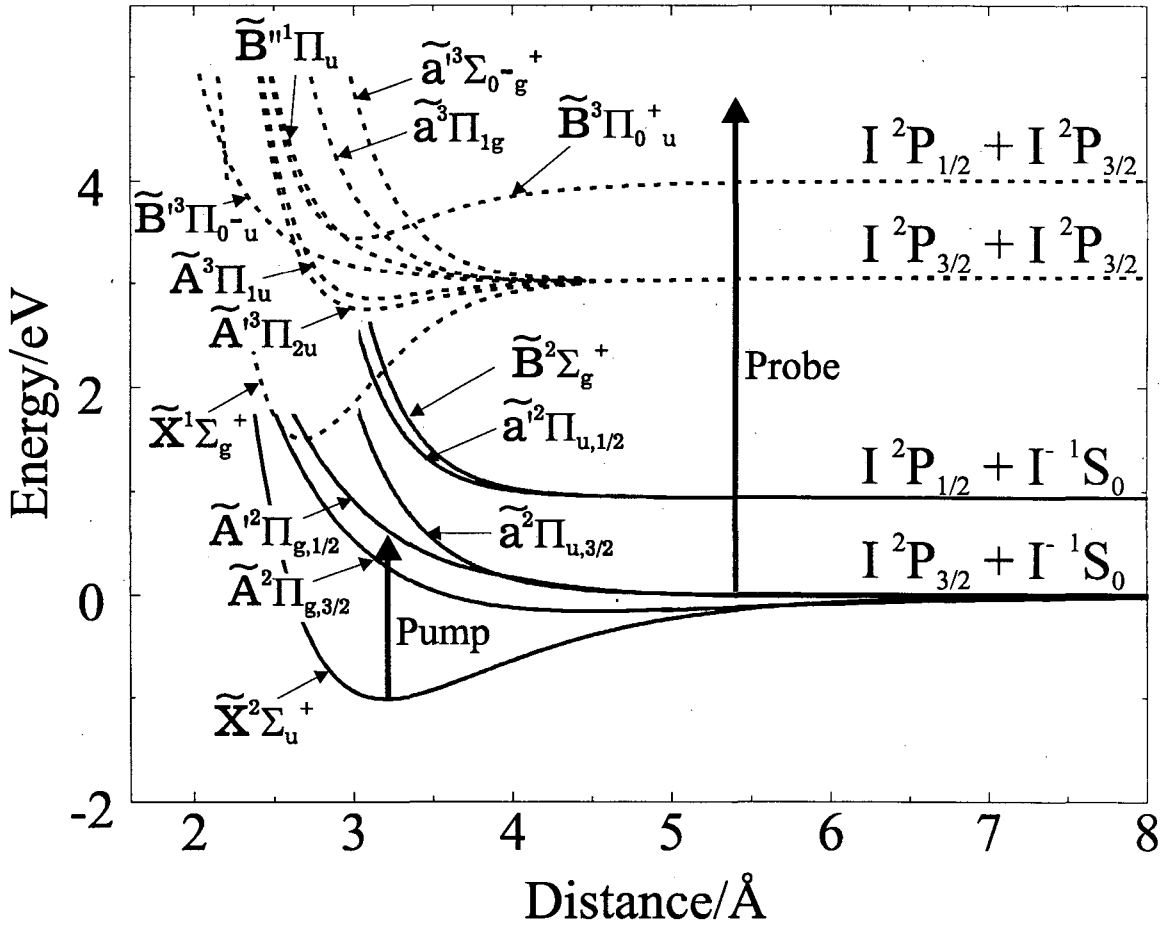


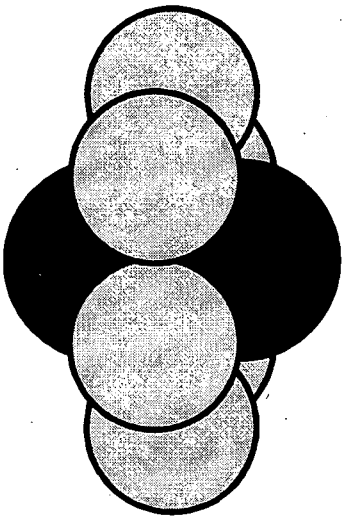
Fig. 1. Potential energy curves for bare I_2^-. Solid lines (--): I_2^-. Dotted lines (·····): I_2.

A time-resolved absorption recovery experiment of the $I_2^-(Ar)_{20}$ cluster was also performed by Vorsa *et al.*,[11] in which cold, mass-selected clusters were excited with a fs-duration pulse at 790 nm, then re-excited with a second, identical pulse after a variable time delay, recording the total flux of two-photon photofragments which presumably indicated the absorption of I_2^- near the bottom of the $\tilde{X}$ state well. This absorption was found to occur with an exponential time constant $t_{1/e}$ of 127 ps. No other transient features were observed, such as those found in large $I_2^-(CO_2)_n$ clusters.[2,9-11] This work was complemented by FPES studies of $I_2^-(Ar)_6$ and $I_2^-(Ar)_{20}$ clusters in our group.[12] In $I_2^-(Ar)_6$, $I^-(Ar)_{n\approx 1}$ was observed to leave the cluster in ~1.2 ps. In $I_2^-(Ar)_{20}$, caging by the solvent resulted in recombination and vibrational relaxation of I_2^- on both the $\tilde{X}$ and $\tilde{A}$ states; these processes were complete in ~200 ps and ~35 ps, respectively.

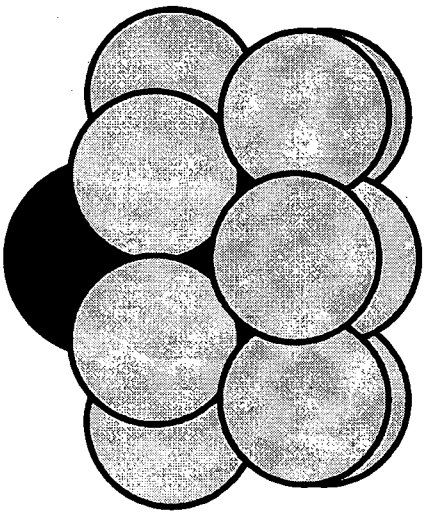
A number of theoretical papers have been published on $I_2^-(Ar)_n$ clusters, exploring both structure and dynamics. Minimum energy structures of $I_2^-(Ar)_n$ clusters have been calculated by Faeder *et al.*[20] and Batista *et al.*[19] In the study of Faeder *et al.*, which appears to be more consistent with experiments,[33] the first 6 Ar atoms surround the I_2^- axis in a ring configuration, with the next 7 Ar atoms solvating one I atom, and any additional atoms cluster to the other I atom, completing a full shell at $n = 20$. Asymmetrically-solvated clusters have an excess negative charge on the more solvated I atom. Fig. 2 shows calculated structures for three cluster sizes: $n = 6$, 12 and 20.

Fig. 2. (On next page) Calculated minimum-energy structures of selected $I_2^-(Ar)_n$ clusters: (a) $I_2^-(Ar)_6$; (b) $I_2^-(Ar)_{12}$; (c) $I_2^-(Ar)_{20}$.

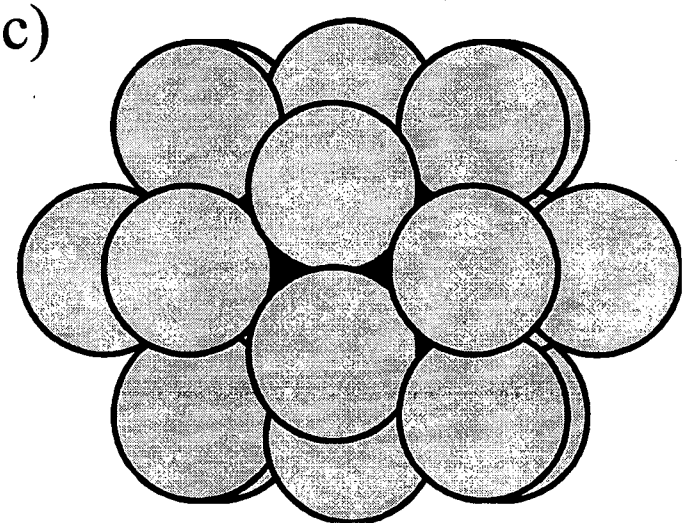
(a)



(b)



(c)



Maslen *et al.*[16] investigated the effect of solvent on charge localization in different I_2^- electronic states. For the two lowest-lying states, the $\tilde{X}$ and $\tilde{A}$ (see Fig. 1), the charge is attracted to the more solvated atom, localizing completely at sufficiently large internuclear distances, a process called "normal charge-switching." In the $\tilde{A}'$ state, however, the polarizability of the molecule is negative along the I_2^- axis,[20] so that the charge tends to localize on the less solvated atom, a process called "anomalous charge-switching." This is illustrated schematically in Fig. 3 for $I_2^-(Ar)_{12}$, an asymmetrically-solvated cluster, adapted from Delaney *et al.*[21] Solid lines indicate potential surfaces near the equilibrium radius of I_2^-, while dotted lines indicate surfaces near the dissociation asymptote. The cluster drawings indicate the solvent configuration and location of the charge. The "solvent coordinate" is defined as the change in energy when the charge is moved to the opposite I atom. When I_2^- is excited by the pump pulse (represented by the thick vertical arrow) from the $\tilde{X}$ to the $\tilde{A}'$ state, the energetically favorable solvent configuration on the $\tilde{X}$ state becomes energetically unfavorable on the $\tilde{A}'$ state. This results in motion of the solvent atoms back toward the charge. However, the solvent atoms are unable to completely surround the charge, because it is always localized on the less solvated I atom, resulting in a symmetric solvent distribution as the minimum energy structure in this state. As the I_2^- bond lengthens, the likelihood of an electronic transition to the $\tilde{X}$ or $\tilde{A}$ state increases; when this occurs, the solvent atoms will rearrange into a more heavily solvated configuration around the I^-, similar to the starting arrangement.

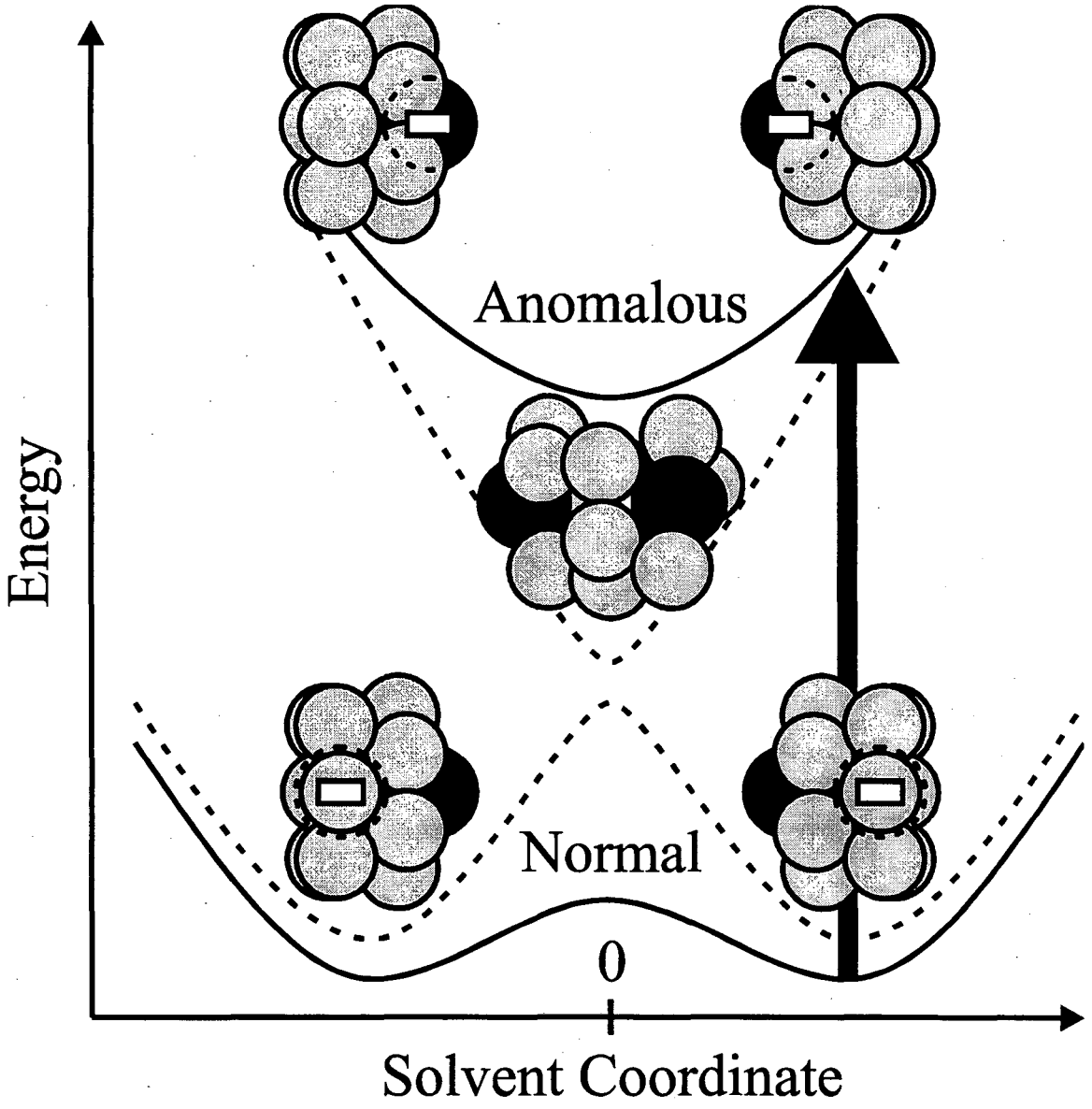


Fig. 3. Illustration of charge-switching. Solid lines indicate potential surface near the equilibrium radius of I_2^-; dotted lines indicate potential surface near the dissociation asymptote. Drawings indicate schematically the solvent configuration and location of the charge. The symmetric configuration (center drawing) has the charge shared equally by both I atoms. The "solvent coordinate" is defined as the change in energy when the charge is moved to the opposite I atom. The thick vertical arrow indicates the $\tilde{A}' \leftarrow \tilde{X}$ excitation. (Adapted from Delaney *et al.*)[21]

Time-resolved dynamics of $I_2^-(Ar)_n$ clusters were investigated by Faeder *et al.*,[20,23] with molecular dynamics simulations, using a surface-hopping algorithm to model electronic transitions. Photoelectron spectra were simulated for $I_2^-(Ar)_6$ and

$I_2^-(Ar)_{20}$ to allow comparison with FPES work from Greenblatt *et al.*[12]. In all clusters, the simulations predicted that I^- and I fragments separated rapidly until ~1 ps; in larger clusters $[I_2^-(Ar)_{n \geq 9}]$,[20,37] some trajectories underwent a transition to the $\tilde{X}$ or $\tilde{A}$ state, resulting both in I^- dissociation, and I_2^- recombination and vibrational relaxation. For $I_2^-(Ar)_{20}$, recombination on the $\tilde{X}$ state occurred in 5-10 ps, with vibrational relaxation requiring more than 200 ps to complete. The number of solvent atoms evaporated tracked the solute internal energy closely. Recombination on the $\tilde{A}$ state took up to 40 ps, but relaxation was much more rapid (~10 ps), owing to the smaller amount (~10%) of internal energy required to be dissipated. However, the solvent evaporation rate was slow, comparable to the $\tilde{X}$ state.

Batista *et al.*[19] also investigated $I_2^-(Ar)_n$ photodissociation using a similar molecular dynamics/surface-hopping algorithm. Their results were much the same as for Faeder *et al.*, reproducing all the asymptotic features observed in Vorsa *et al.* Recombination on both the $\tilde{X}$ and $\tilde{A}$ states was increasingly rapid as cluster size increased, from ~10 ps in $I_2^-(Ar)_{11}$, to ~3 ps in $I_2^-(Ar)_{19}$, but solvent evaporation in the largest cluster took longer than the time scale of the simulations, 45 ps. They also observed a larger number of solvent atoms in the I_2^- $\tilde{A}$ state at long time delays than in Vorsa *et al.*,[6,7] implying a slow evaporation rate from this state.

The FPES experiment excites a mass-selected anion cluster with a femtosecond pump pulse from the $\tilde{X}$ state to the dissociative $\tilde{A}'$ state. A second, delayed femtosecond probe pulse generates a photoelectron spectrum. Features arising from I^-, and the I_2^- $\tilde{X}$ state in different vibrational levels, are readily distinguished. The I_2^- $\tilde{A}$

state appears similar to I^-, but is still discernible. The number of solvent atoms surrounding the anion in each state may also be determined from the spectrum.

The major goal of this study, as well as the accompanying study of $I_2^-(CO_2)_n$ clusters, is to use FPES to observe how the dynamics evolve from the uncaged to caged cluster size limits. Key findings for $I_2^-(Ar)_n$ clusters include: 1). Determination of the initial solvent configuration from measuring the number of solvent atoms around I^- at early time delays (~300 fs). This confirms the anomalous charge-switching predictions of Maslen *et al.*,[16] where the electron hops to the less-solvated I atom upon excitation to the $\tilde{A}'$ state. 2). Measurement of the time-resolved number of solvent atoms in both the I^- and I_2^- $\tilde{X}$ state channels, providing information on relaxation dynamics from the point of view of solvent evaporation. This enabled the time evolution of uncaged fragments to be observed, and the rates of solvent loss and vibrational relaxation to be compared in caged clusters, such as $I_2^-(Ar)_{20}$. 3). A detailed picture of the vibrational relaxation in caged photofragments, especially $I_2^-(Ar)_{20}$, which relaxes almost completely over a ~200 ps timescale. 4). Unambiguous identification of the I_2^- $\tilde{A}$ state, along with information about its time evolution, which provided a more complete characterization of this state.

2. Experimental

The experimental apparatus has been described in considerable detail elsewhere,[13] and will only be summarized briefly here. To generate cluster anions, Ar carrier gas (20 psig) is passed over solid I_2 and expanded into vacuum through a piezoelectric pulsed valve running at a repetition rate of 500 Hz. A 1.5 keV electron gun crosses the resulting supersonic expansion, creating vibrationally cold negative ions, which are then pulse-extracted into a Wiley-McLaren[38] time-of-flight mass spectrometer. Femtosecond pump

(780 nm, 80 fs, 150 μJ) and probe (260 nm, 100 fs, 20 μJ) pulses, produced from a Clark-MXR regeneratively amplified Ti:sapphire laser, intersect the ions at the focus of a magnetic bottle electron spectrometer,[39] detaching photoelectrons. Electron kinetic energies (eKE) for the resulting photoelectrons are measured by time-of-flight. High collection efficiency of the magnetic bottle enables rapid acquisition (400-1200 s) of photoelectron spectra. Since the probe photon has sufficient energy to detach electrons from the ground state of $I_2^-(Ar)_n$ clusters, spectra are not background-free, so a fraction of this "probe only" spectrum was subtracted from the pump-probe spectra in order to facilitate observation of the two-photon signals. The energy resolution of the I_2^- photoelectron spectrum has been improved ~4× using pulsed deceleration[40] of the anions just prior to laser interaction. This technique, recently added to the spectrometer, was only employed for bare I_2^-. However, since the resolution scales as $(EU/m)^{1/2}$, where E is the electron kinetic energy, U is the anion kinetic energy, and m is the anion mass,[36] the heavier $I_2^-(Ar)_n$ clusters have an inherently narrower resolution, and light clusters were measured at slower beam energies to improve their resolution. Typical resolution for 1 eV electrons was 90 meV for I_2^-, and 90-130 meV for $I_2^-(Ar)_n$ clusters.

3. Results

Time-resolved photoelectron spectra have been measured for I_2^- and for $I_2^-(Ar)_n$ clusters with n = 6, 9, 12, 16 and 20. Each molecule was studied at several pump-probe time delays, over a range of ~50-200 ps. In addition, the $I_2^-(Ar)_{20}$ cluster was measured at 3 ns pump-probe delay. Spectra at selected time delays are shown in Figs. 4-5. Features have been labeled with capital letter designations, following a scheme summarized in Table 1 which is consistent among all the clusters studies, as well as with the $I_2^-(CO_2)_n$

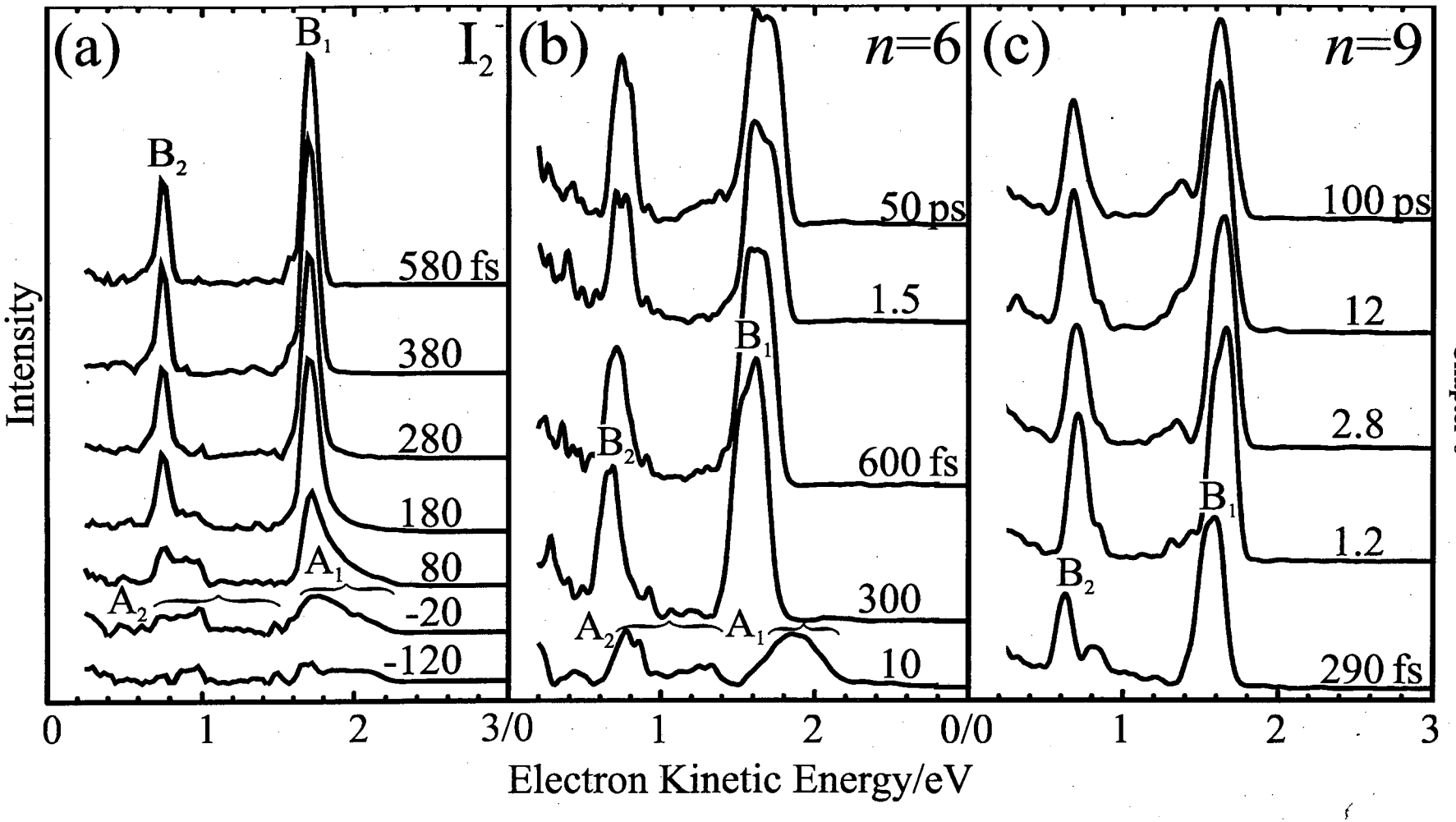
clusters in the accompanying paper,[35] to allow for comparisons across cluster size and type. These assignments are based on previous work,[12,13,36] and the analysis presented in the following section.

Bare I_2^- displays two broad features A_1 (1.7-2.2 eV) and A_2 (0.8-1.3 eV) peaking near 0 fs, transforming into sharper features B_1(1.71 eV) and B_2 (770 meV) which reach full height by 380 fs. The A features originate from wavepacket overlap with neutral potential energy curves in the initial Franck-Condon region[36,41] (see Fig. 1). The B features, differing in energy by the spin-orbit splitting of neutral I (943 meV),[42] correspond to fully dissociated I^-.

Table 1. Labeling system of features observed in FPES, with corresponding assignments. In cases where two spin-orbit manifolds are visible (peaks A, B and D), each is labeled with a subscript, e.g. A_1 and A_2, according to decreasing eKE.

Label	Assignment
A_1, A_2	$I_2 \leftarrow I_2^- \ \tilde{A}'$ (short-time transient)
B_1, B_2	$I \leftarrow I^-$
D_1, D_2	$I_2 \leftarrow I_2^- \ \tilde{A}$
E	$I_2 \ \tilde{X} \leftarrow I_2^- \ \tilde{X}$ inner turning point (ITP)
F	$I_2 \ \tilde{X} \leftarrow I_2^- \ \tilde{X}$ outer turning point (OTP),
	$I_2^* \leftarrow I_2^- \ \tilde{X}$

Fig. 4. (On next page) FPES at selected time delays: (a) I_2^-; (b) $I_2^-(Ar)_6$; (c) $I_2^-(Ar)_9$. Pump photon energy = 1.589 eV, probe photon energy = 4.768 eV.

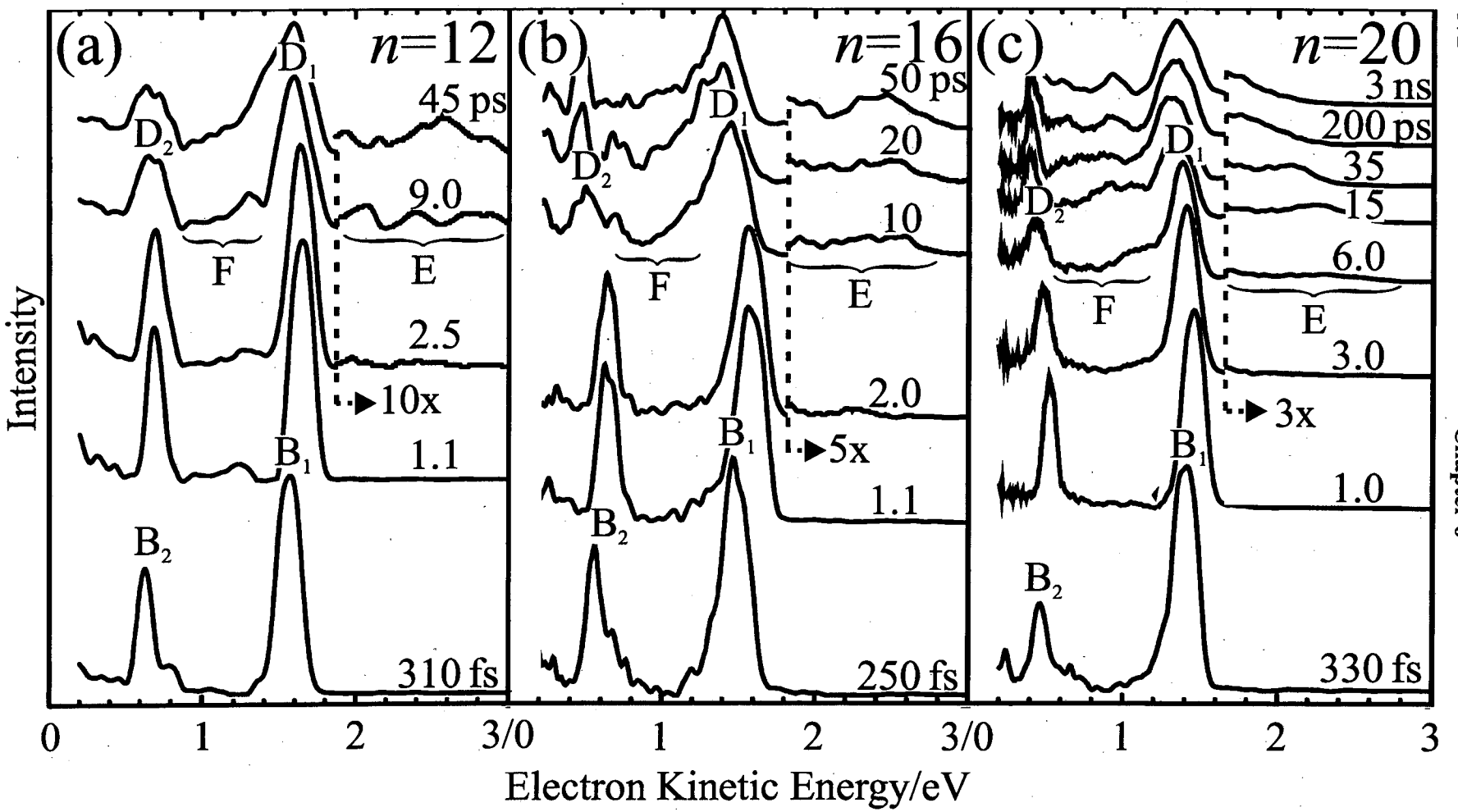

(a)
I_2^-
(b) n=6
(c) n=9
B_1
B_2
A_1
A_2
580 fs
380
280
180
80
-20
-120
50 ps
1.5
600 fs
300
10
100 ps
12
2.8
1.2
290 fs
Intensity
0
1
2
3/0
1
2
0/0
1
2
3
Electron Kinetic Energy/eV

In $I_2^-(Ar)_6$, the spectrum initially resembles bare I_2^-, displaying A features at 10 fs which evolve to B features by 300 fs. B_1 and B_2 are shifted 120 meV to lower eKE relative to bare I^-. This shift is due to the presence of Ar atoms, since the I^--Ar bond is stronger than that of I-Ar, resulting in an increase in electron affinity. Between 300 fs and 1.5 ps, the energies of the B features increase 80 meV, indicating a lessening interaction between the Ar atoms and I^-, as the I^- fragment separates from the cluster. Through 200 ps there is an additional eKE increase of 10 meV, due to Ar atom evaporation.

For $I_2^-(Ar)_9$, the B features appear 150 meV lower than bare I^-, and from 300 fs to 1.1 ps their eKE's increase 80 meV, as for $I_2^-(Ar)_6$. However, from 2.8 to 12 ps, the features decrease 30 meV in energy, indicating an increased number of Ar atoms around I^-, the cause for which will be explored in the Discussion section. Through 100 ps, the energy increases again by 10 meV, due to Ar atom evaporation.

In the $I_2^-(Ar)_{12}$ cluster, the B features closely track those in $I_2^-(Ar)_9$ from 310 fs through 2.5 ps, after which they broaden significantly toward lower eKE, and decrease ~25% in integrated intensity. At this point (9.0 ps), the features are relabeled D_1 and D_2. The broadening is attributed to partial recombination on the I_2^- $\tilde{A}$ state (see Discussion). Between 2.5 and 9.0 ps, feature E appears between 1.9 and 3.0 eV, along with a broad feature F between 900 meV and 1.3 eV. These features are assigned to vibrationally excited I_2^- on the $\tilde{X}$ state. Both features E and F become more prominent out to 45 ps.

Fig. 5. (On next page) FPES at selected time delays: (a) $I_2^-(Ar)_{12}$; (b) $I_2^-(Ar)_{16}$; (c) $I_2^-(Ar)_{20}$. Pump photon energy = 1.589 eV, probe photon energy = 4.768 eV.


(a)
n=12
(b)
n=16
(c)
n=20
D1
D2
F
E
B1
B2
45 ps
9.0
2.5
10x
1.1
310 fs
50 ps
20
10
2.0
5x
1.1
250 fs
3 ns
200 ps
35
15
6.0
3.0
3x
1.0
330 fs
Intensity
0
1
2
3/0
1
2
3/0
1
2
3
Electron Kinetic Energy/eV

The $I_2^-(Ar)_{16}$ spectra display B features initially 220 meV lower than I^-, increasing 90 meV through 1.1 ps. Between 2.0 and 20 ps the features broaden and shift to lower eKE by 240 meV, where they are relabeled D_1 and D_2, due to recombination on the $\tilde{A}$ state. Between 2.0 and 10 ps, features E (1.7 and 2.9 eV) and F (0.8-1.3 eV) grow in, more intense relative to the D features than in $I_2^-(Ar)_{12}$, due to recombination on the I_2^- $\tilde{X}$ state. Between 10 and 50 ps, the high-energy edge of feature E shifts ~100 meV to lower energy, indicating partial vibrational relaxation.

For $I_2^-(Ar)_{20}$, the B features appear 300 meV lower than I^- and increase 50 meV through 1.0 ps. After this time delay, they reverse direction and are relabeled D_1 and D_2, shifting 140 meV to lower eKE through 35 ps, due to $\tilde{A}$ state recombination, with broadening and a ~40% decrease in integrated intensity. Features E (1.6-2.7 eV) and F (0.5-1.2 eV) appear by 6.0 ps. The high-energy edge of feature E shifts ~700 meV to lower eKE through 3 ns, while feature F undergoes a complex evolution in structure. The changes in features E and F are due to extensive $\tilde{X}$ state vibrational relaxation.

4. Analysis

The goal in simulating the FPES spectra is to determine, at each time delay, the state of the cluster. This consists of answering four basic questions: 1). Have I and I^- recombined? 2). If I^- is present, how many solvent molecules surround it? 3). If I_2^- is present, what is the electronic and vibrational state, and how many solvent molecules surround it? 4). What are the relative populations of the different states (I_2^- $\tilde{X}$, I_2^- $\tilde{A}$, I^-) in the cluster? In certain cases, these questions can be answered immediately by looking at the spectrum, whereas others require iterative refinement of simulation parameters to accurately characterize the cluster. Simulations were performed using a combination of

measured spectra and theoretical calculations, the procedures for which are detailed below. Unlike previous studies of bare I_2^-,[36,41] no explicitly time-dependent calculations were performed, as the number of degrees of freedom in $I_2^-(Ar)_n$ clusters were far too large for our existing wavepacket propagation programs.

It was assumed that, after the initial (~300 fs) I_2^- dissociation, I and I^- are well-separated, so that I has little influence on the photoelectron spectrum of I^-. This is substantiated by the observation that pairs of features (B_1 and B_2) are present in all spectra at short times (< 1-2 ps), differing in energy by approximately the spin-orbit splitting of I (943 meV) which is characteristic of the photoelectron spectrum of I^-, though shifted to lower eKE. The shift is a well-understood effect, arising from the difference in binding energy between the I^--Ar (45.8 meV) and I-Ar (18.8 meV) bonds.[43] These differences have been measured precisely using zero electron kinetic energy (ZEKE) and partially-discriminated threshold photodetachment spectroscopy of $I^-(Ar)_n$ clusters.[43]

The initial average number of Ar atoms around I^- ("$<n_{I^-}>$") were calculated by comparing the eKE of feature B_1 at ~300 fs to the measured values in the above study, using linear interpolation to estimate a fractional $<n_{I^-}>$ when the energy lay between measured shifts. Results are presented in Table 2, along with estimates of $<n_{I^-}>$ for the anomalous and normal charge-switching states, based on calculated structures,[20] and assuming no loss of Ar atoms. For analyzing $<n_{I^-}>$ at later time delays, Fig. 6 shows the eKE of feature B_1 *vs.* time for all clusters, with $<n_{I^-}>$ indicated on the righthand side of the figure. For $I_2^-(Ar)_{n \geq 12}$, the graph stops when the feature begins to decrease in energy and is relabeled D_1, as it no longer reflects a pure I^- signal.

Table 2. Number of solvent atoms $\langle n_I^- \rangle$ for feature B_1 at ~300 fs, along with estimated $\langle n_I^- \rangle$ for anomalous and normal charge-switching states, calculated from model structures.[23]

Parent cluster	B_1	Anomalous	Normal
6	4.7	6.0	6.0
9	5.4	6.0	9.0
12	5.8	6.0	12.0
16	9.1	9.0	13.0
20	13.3	13.0	13.0

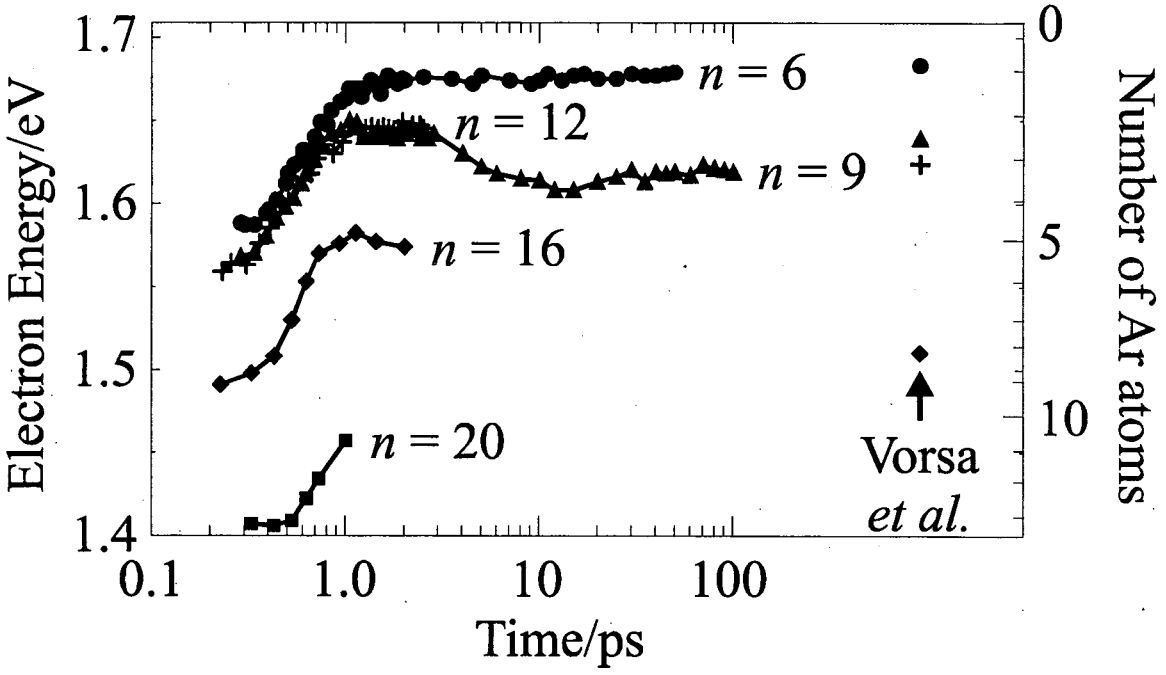


Fig. 6. Center eKE of feature B_1 *vs.* time, for all $I_2^-(Ar)_n$ clusters. Number of Ar atoms ($\langle n_I^- \rangle$) is shown on righthand axis, calculated from Yourshaw *et al.*[43]

For simulating the FPES spectra, $I^-(Ar)_n$ features were generated from a measured probe-only spectrum of bare I^-, shifted in energy according to the known solvent shift. The integrated intensity of the $I\ ^2P_{3/2} \leftarrow I^-\ ^1S_0$ transition was taken to be 2.0 (see Table 3), relative to 1.0 for the $I_2\ \tilde{X} \leftarrow I_2^-\ \tilde{X}$ ($v = 0$) transition as determined by comparing the integrated intensities of I_2^- bleach and I^- (signal) features in the FPES of bare I_2^-. The intensity of the $I\ ^2P_{1/2} \leftarrow I^-\ ^1S_0$ transition was empirically determined to be 0.6. The final

spectrum was convoluted with an instrument resolution function,[36] calibrated approximately for experimental conditions.

Table 3. Relative integrated intensities of transitions used in simulated spectra.

Transition	Relative integrated intensity
$I_2\ \tilde{X} \leftarrow I_2^-\ \tilde{X}$ (v = 0)	1.0
$I_2\ \tilde{X} \leftarrow I_2^-\ \tilde{X}$ (v > 0)	0.4-1.6*
$I_2\ \tilde{A}' / \tilde{A} \leftarrow I_2^-\ \tilde{X}$	0.24
$I_2\ \tilde{B}' / \tilde{B}'' \leftarrow I_2^-\ \tilde{X}$	0.2
$I_2\ \tilde{a} \leftarrow I_2^-\ \tilde{X}$	0.15-0.5**
$I_2\ \tilde{a}' \leftarrow I_2^-\ \tilde{X}$	1.5× ($I_2\ \tilde{a} \leftarrow I_2^-\ \tilde{X}$)
$I_2\ \tilde{B} \leftarrow I_2^-\ \tilde{X}$	0.5-1.0**
$I_2\ \tilde{A}' / \tilde{A} / \tilde{B}' / \tilde{B}'' \leftarrow I_2^-\ \tilde{A}$	0.22
$I_2\ \tilde{a} \leftarrow I_2^-\ \tilde{A}$	0.44
$I_2\ \tilde{a}' \leftarrow I_2^-\ \tilde{A}$	0.66
$I_2\ \tilde{B} \leftarrow I_2^-\ \tilde{A}$	0.6
$I\ ^2P_{3/2} \leftarrow I^-\ ^1S_0$	2.0
$I\ ^2P_{1/2} \leftarrow I^-\ ^1S_0$	0.6

*Spectra scaled with energy-dependent function $f(E)$; see below and text.
**Varied with spectrum.

To simulate the photoelectron spectrum arising from a vibrationally excited $I_2^-\ \tilde{X}$ state, wavefunctions were calculated using standard procedures for a Morse oscillator,[44] and a photoelectron spectrum for each vibrational level was generated using a time-dependent propagation method[45] to calculate the Franck-Condon overlap with various neutral states. The $I_2^-\ \tilde{X}$ state potential parameters are identical to Ref. 34. Neutral state parameters are identical to Ref. 41, with the exception of the $\tilde{B}'$ state, whose repulsive wall was adjusted empirically to fit a high-resolution (~10 meV) photoelectron spectrum of I_2^-.[33] It was assumed that spectra arose from an incoherent superposition of vibrational levels, so composite spectra were constructed by summing spectra from individual vibrational wavefunctions over a distribution of levels. The presence of Ar atoms

decreases the eKE of I_2^- features, much as for $I^-(Ar)_n$. These solvent shifts have been measured for vibrationally cold $I_2^-(Ar)_n$ clusters using photoelectron spectroscopy,[33] and are used to shift the simulated spectra. The shifts are smaller than for $I^-(Ar)_n$, despite the fact that the I_2^--Ar binding energy (53 meV) is larger than that of I^--Ar (45.8 meV);[43] this is due to a somewhat larger I_2-Ar binding energy over that of I-Ar. It is assumed that the shifts do not change for $v > 0$. Note that the I_2^--Ar bond energy is significantly lower than the energy lost per Ar atom due to evaporation (73 meV), as calculated in Vorsa *et al.* from the number of solvent atoms remaining in I_2^- fragments from large ($n > 20$) parent clusters. This discrepancy was attributed to kinetic energy of the departing Ar atom.[7]

Two simulated photoelectron spectra of the I_2^- $\tilde{X}$ state and their relation to the I_2 potential energy curves are shown in Fig. 7 for (a) $v = 0$ and (b) $v = 20$. The vibrationally cold spectrum ($v = 0$) consists of an extended progression, unresolvable with instrument resolution, centered at 1.54 eV (the I_2 $\tilde{X}$ state), a pair of narrow features at 1.00 eV [$\tilde{A}'(^3\Pi_{2u})$] and 900 meV [$\tilde{A}(^3\Pi_{1u})$], another pair of narrow features at 650 meV [$\tilde{B}'(^3\Pi_{0^-u})$] and 540 meV [$\tilde{B}''(^1\Pi_u)$], and two broad, overlapping features at ~300 meV [$\tilde{a}(^3\Pi_{1g})$ and $\tilde{B}(^3\Pi_{0^+u})$]. The $\tilde{a}'(^3\Sigma^-_{0^+g})$ state is not observable, as it is not accessible by the probe photon from I_2^- $\tilde{X}$ ($v = 0$).

For the vibrationally excited $\tilde{X}$ state ($v = 20$), the shapes and energies of the photoelectron features change considerably. Since the amplitude of the $\tilde{X}$ state wavefunction is concentrated near the classical inner and outer turning points of the potential (ITP and OTP, respectively), Franck-Condon overlap with I_2 states will be largest in these regions. For the I_2 $\tilde{X}$ $\leftarrow$ I_2^- $\tilde{X}$ transition, the large change in I_2 $\tilde{X}$

potential energy with internuclear distance produces distinctive and well-separated features arising from each region (see Fig. 7): an extended tail at high eKE, arising from the ITP region, and a narrower, intense peak at low eKE due to the OTP region. The ITP region of the spectrum is very sensitive to ν, whereas the OTP region is fairly

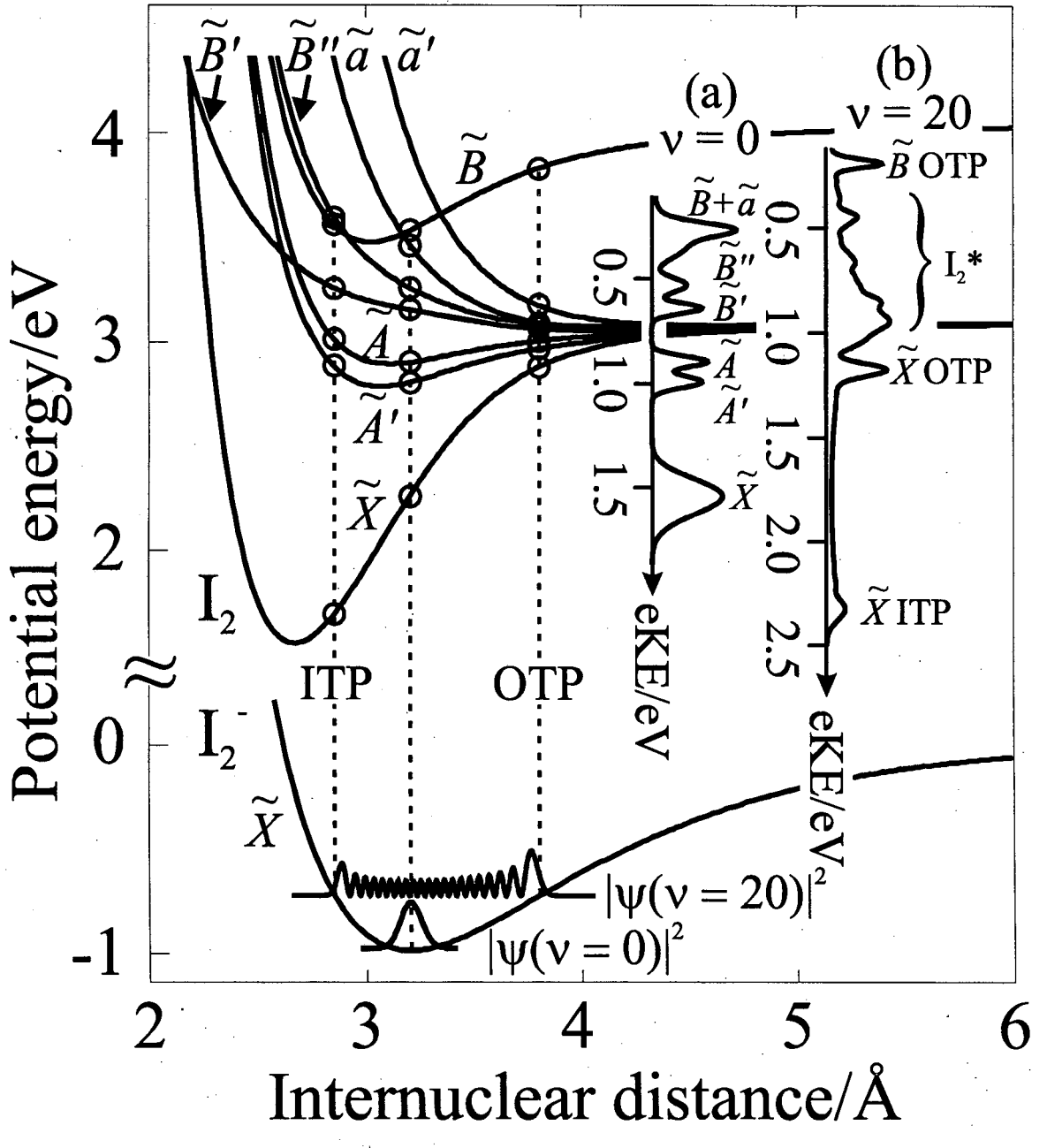


Fig. 7. Example of how I_2^- $\tilde{X}$ state wavefunctions in different vibrational levels give rise to very different photoelectron spectra. (a) ν = 0; (b) ν = 20.

independent of v over a wide range (~10-30). At higher vibrational levels, the OTP energy increases with v. For the $\tilde{A}' \leftarrow \tilde{X}$ and $\tilde{A} \leftarrow \tilde{X}$ transitions, the difference in eKE between the ITP and OTP regions is much less, though there is a considerable broadening for v > ~30. The $\tilde{B}'$, $\tilde{B}''$, $\tilde{a}$ and $\tilde{a}'$ states display a wider range of potential energies, so that the OTP regions of these transitions overlap with the $\tilde{A}' / \tilde{A} \leftarrow \tilde{X}$, and the ITP regions occur at much lower eKE. The $\tilde{B} \leftarrow \tilde{X}$ transition, correlating at large internuclear distance with the $I\ ^2P_{1/2} \leftarrow I^-\ ^1S_0$ transition, appears near 200 meV for $v \leq 60$.

In order to determine <v> from a spectrum, the number of solvent atoms ("$\langle n_X \rangle$") must also be known, since both parameters affect the eKE of features E and F. When <v> is very small (< ~5), the $\tilde{X} \leftarrow \tilde{X}$ transition is compact and the shape depends sensitively on <v>, so both $\langle n_X \rangle$ and <v> may be simultaneously determined by simulating the shape and energy of feature E. For larger <v>, the $\tilde{X} \leftarrow \tilde{X}$ ITP energy is mostly governed by <v>, but $\langle n_X \rangle$ strongly modifies it, so feature E cannot be used to determine <v> exclusively. However, feature F, arising from the $\tilde{X} \leftarrow \tilde{X}$ OTP and $\tilde{A}' / \tilde{A} / \tilde{B}' / \tilde{B}'' / \tilde{a} / \tilde{a}' / \tilde{B}$ (collectively referred to as I_2^*) $\leftarrow \tilde{X}$ transitions, is more sensitive to $\langle n_X \rangle$ than to <v>, and for <v> between ~5 and ~30, two distinct peaks are visible which can be used in conjuction with the $\tilde{X} \leftarrow \tilde{X}$ ITP transition to obtain both <v> and $\langle n_X \rangle$. For <v> larger than ~30, the $\tilde{X} \leftarrow \tilde{X}$ OTP and $I_2^* \leftarrow \tilde{X}$ transitions coalesce into a single, broad peak, and determination of $\langle n_X \rangle$ is less precise.

The integrated intensities of transitions from I_2^- $\tilde{X}$ to different I_2 electronic states were empirically determined from fitting a one-photon spectrum of bare I_2^-, normalizing the $\tilde{X} \leftarrow \tilde{X}$ (v = 0) transition to 1.0 as a reference (see Table 3). It is assumed that these

intensities do not change in I_2^- clusters. For $v > 0$, it has been shown in the course of fitting spectra that the $\tilde{X} \leftarrow \tilde{X}$ transition dipole moment varies with eKE as well as v, the most dramatic deviations being observed in the ITP and OTP regions at large v (> ~30), with apparent intensities 1.6 and 0.4 times the $v = 0$ intensities, respectively. This is not unexpected, since the wavefunctions are quite extended for vibrationally excited levels, and the overlap of the electronic orbitals will be significantly different than at the equilibrium bond distance, changing the relative cross section. Therefore, to obtain the best estimate of the true integrated intensities, a smooth, energy-dependent scaling function $f(E)$ was applied to the simulated spectra for the $\tilde{X} \leftarrow \tilde{X}$ transition:

$$f(E) = \frac{a_1 - 1}{1 + e^{-(E-E_1)/k_1}} + \frac{a_2 - 1}{1 + e^{-(E-E_2)/k_2}} + 1 \qquad (1)$$

where E is electron kinetic energy (eV) before applying any solvent shifts. Parameters for this function are summarized in Table 4. Note that a_1, which governs the $\tilde{X} \leftarrow \tilde{X}$ OTP intensity, was varied in different spectra from 0.4 to 1.0, following an inverse trend with <v>, which indicated a decreasing transition dipole moment as the internuclear radius increased. No such modifications were made to the simulated spectra for transitions to higher-lying states, with the exceptions of the $\tilde{a}/\tilde{a}' \leftarrow \tilde{X}$ and $\tilde{B} \leftarrow \tilde{X}$ transitions. Here, the relative intensity of the $\tilde{a} \leftarrow \tilde{X}$ transition was freely varied to best fit feature F, with the $\tilde{a}' \leftarrow \tilde{X}$ intensity scaled to 1.5× that of the $\tilde{A} \leftarrow \tilde{X}$, in accord with the estimated relative intensities of these transitions, as discussed in Zanni *et al.*[41] The scaling of the $\tilde{B} \leftarrow \tilde{X}$ transition, which appeared at very low eKE in all spectra, was varied between 0.5 and 1.0 to best fit this region. The final spectrum was convoluted with the same instrument resolution function as for I^- above.

Table 4. Parameters used in $f(E)$ function for scaling I_2 $\tilde{X}$ ← I_2^- $\tilde{X}$ transition. Equation is defined in the text.

Parameter	Value
A1	0.4-1.0*
A2	1.6
k_1	-0.02 eV
k_2	0.08 eV
E_1	1.3 eV
E_2	2.0 eV

*Varied with <v>.

Simulation of the I_2^- $\tilde{A}$ state was considerably less precise than the $\tilde{X}$ state. A simple Morse function was employed, using parameters (R_e = 4.7 Å, D_e = 140 meV) slightly modified from Greenblatt *et al.*[12] to better fit the $I_2^-(Ar)_{20}$ spectrum at 3 ns, assuming that the number of solvent atoms ("$<n_A>$") was equal to the photofragmentation average for this cluster (11.1).[6] This assumption was supported by the lack of any significant change in the energy of the $\tilde{A}$ state features (D_1 and D_2) between 15 ps and 3 ns, indicating an asymptotic solvent configuration had been achieved; see the Discussion for more details. Transitions from v = 0 to each neutral state (except $\tilde{X}$) were weighted equally, as done in Zanni *et al.* for the I_2^- $\tilde{A}'$ state.[41] The broadening of feature D_1 is likely due to the repulsive regions of the I_2 states ($\tilde{B}'$, $\tilde{B}''$, $\tilde{a}$, $\tilde{a}'$), but as these are poorly defined, adequate reproduction of this broadening was not possible. To simulate the broad appearance, therefore, this feature was convoluted with a wide resolution function (~250 meV). $<n_A>$ was determined from the experimental spectra using the energy shifts of the $I^-(Ar)_n$ clusters, rather than the $I_2^-(Ar)_n$ clusters, based on the assumption that the electron is localized on a single I atom at the large equilibrium bond distance, and is therefore stabilized by solvent to the same extent as for I^-. This

assumption was borne out in other clusters by fairly good agreement between the calculated $\langle n_A \rangle$'s at long time delays and the photofragmentation averages (see Discussion). The integrated intensity was assumed to be the same as for I^-, which was also supported experimentally.

Populations of the I^-, I_2^- $\tilde{X}$ and I_2^- $\tilde{A}$ contributions, indicated by "P_I-," "P_X," and "P_A," respectively, were determined from the intensities of simulated spectral features, weighted by their relative cross-sections. Populations sum to unity for all spectra.

Numerous time delays have been simulated to follow the dynamics in clusters of $I_2^-(Ar)_{n \geq 12}$. For $I_2^-(Ar)_{12\text{-}16}$, where changes are minimal once features E and F have appeared, only a single, long time delay (45-50 ps) is shown in Figs. 8(a-b). In $I_2^-(Ar)_{20}$, where significant evolution is observed in the spectra after the appearance of these features, several time delays (6.0, 15, 35 and 200 ps) are shown in Figs. 8(c-f). Each figure includes curves representing the I^- [for $I_2^-(Ar)_{12}$ only], I_2^- $\tilde{X}$ and I_2^- $\tilde{A}$ contributions, the total simulated spectrum, and the experimental spectrum. Simulation parameters are summarized in Table 5. In Fig. 9, P_X is plotted *vs.* time for all three clusters, for many more time delays than shown in Fig. 8. Photofragmentation values are indicated as detached points on the righthand side of the graph. As $I_2^-(Ar)_{20}$ displays considerable vibrational relaxation not seen in the smaller clusters, Fig. 10(a-b) plots $\langle v \rangle$ and $\langle n_X \rangle$ *vs.* time for this cluster, along with model data from Faeder *et al.*[23] Fig. 10(c) plots a derived quantity $E_{cluster}$ which is defined in the section on $I_2^-(Ar)_{20}$ in the Discussion.

Table 5. Parameters used in simulating spectra of $I_2^-(Ar)_n$ clusters at selected time delays. P_X, P_A and P_{I^-} indicate populations of I_2^- $\tilde{X}$, I_2^- $\tilde{A}$ and I^- fragments, respectively. <v> indicates the average vibrational level of the I_2^- $\tilde{X}$ state. $\langle n_X \rangle$, $\langle n_A \rangle$ and $\langle n_{I^-} \rangle$ indicate the average numbers of Ar atoms surrounding the I_2^- $\tilde{X}$, I_2^- $\tilde{A}$ and I^- fragments, respectively. "MS" indicates results of photofragment experiments from Vorsa *et al.*[6,7]

Parent Cluster	Time (ps)	Population P_X	P_A	P_{I^-}	I_2^- $\tilde{X}$ <v>	Number of Ar $\langle n_X \rangle$	$\langle n_A \rangle$	$\langle n_{I^-} \rangle$
12	45	0.30	0.21	0.49	68.0	0	5.0	3.2
	MS	**0.23**	**0.23**	**0.54**	-	**0**	**2.3**	**3.2**
16	50	0.50	0.50	0	34.4	0	8.0	-
	MS	**0.43**	**0.55**	**0.02**	-	**0**	**6.2**	**8.5**
20	6.0	0.36	0.64	0	40.0	8.0	9.0	-
	15	0.50	0.50	0	29.1	6.0	11.0	-
	35	0.50	0.50	0	14.2	3.0	11.5	-
	200	0.54	0.46	0	5.6	0.5	11.0	-
	MS	**0.44**	**0.56**	**0**	-	**0.2**	**11.1**	-

5. Discussion

This section of the paper is divided into four parts. In the first section, the spectra of dissociated I^- at ~300 fs are examined for all clusters, in order to determine the initial configuration of solvent atoms around the I^-. Then, $I_2^-(Ar)_6$ and $I_2^-(Ar)_9$ clusters are discussed, which display only I^- dynamics. This is followed by a discussion of $I_2^-(Ar)_{12}$ and $I_2^-(Ar)_{16}$ clusters, which display I_2^- $\tilde{A}$ state and I_2^- $\tilde{X}$ state features, but limited vibrational relaxation in the $\tilde{X}$ state. Finally, $I_2^-(Ar)_{20}$ is examined separately, whose spectra display extensive $\tilde{X}$ state vibrational relaxation, in addition to the dynamics observed in smaller clusters. By way of orientation, the fraction of I_2^- products measured by Vorsa *et al.*[6] is useful to state here: 0.00 (n = 6 and 9), 0.46 (n = 12), 0.98 (n = 16), 1.00 (n = 20). Roughly equal amounts of low- and high-mass fragments are observed in the I_2^- products, which will be shown below to correspond to the I_2^- $\tilde{X}$ and $\tilde{A}$ states, respectively.

5.1. Short-time dynamics

An unverified prediction of Maslen *et al.*[16] is whether the electron localizes on the less solvated I atom after excitation to the $\tilde{A}'$ state. When I^- first appears at ~300 fs after photodissociation in bare I_2^-, the distance between I^- and I is not very large (6.4 Å).[23] Therefore, the number of Ar atoms $\langle n_{I^-} \rangle$ surrounding I^- has probably not changed much from the initial configuration around I_2^-. In Table 2, we examine this measured quantity, along with estimated numbers assuming the charge is localized on the more- and less-solvated I atom (corresponding to the normal and anomalous charge-switching states, respectively). For $I_2^-(Ar)_{9\text{-}16}$, where the difference between the two estimates is 3.0-6.0 Ar atoms, the observed $\langle n_{I^-} \rangle$'s are in excellent agreement (0.1-0.6) with the anomalous charge-switching estimate. In the symmetrically-solvated $I_2^-(Ar)_6$ and $I_2^-(Ar)_{20}$ clusters, where there is no distinction between the normal and anomalous configurations, $\langle n_{I^-} \rangle$ is still fairly close to the estimated number, though for $I_2^-(Ar)_6$ it is significantly lower (1.3) than the estimate. This discrepancy probably arises from the virtually unimpeded motion of I^- away from the Ar atoms, lowering the apparent average, whereas in $I_2^-(Ar)_{20}$, with I^- motion more arrested by solvent,[23] the discrepancy is expected to be less, as verified by the excellent agreement (0.3) with the model for this cluster.

5.2. $I_2^-(Ar)_6$ and $I_2^-(Ar)_9$

For these two clusters, the increase in the eKE of the B features after ~300 fs results from a decreasing number of solvent atoms surrounding I^-; from Fig. 6, it is seen that $\langle n_{I^-} \rangle$ decreases until ~1.5 ps. One possible explanation for the decrease is that the loss of solvent is due to fast ejection of neutral I, leaving behind a vibrationally excited $I^-(Ar)_n$ cluster which evaporates Ar atoms until the available energy is dissipated. This

mechanism was suggested in studies of $Br_2^-(CO_2)_n$ clusters.[14] However, theoretical simulations by Faeder *et al.*[20,23] predict that, in small clusters such as $I_2^-(Ar)_6$, the I^- fragment simply leaves the cluster, capturing one or more Ar atoms during its escape. Hence, as discussed previously,[12] the decrease in $\langle n_{I^-} \rangle$ reflects the steady weakening of the attractive interaction between the I^- and Ar solvent atoms.

For $I_2^-(Ar)_6$, changes to the spectrum are essentially over by 1.5 ps, at which point the value of $\langle n_{I^-} \rangle$ (1.2), is close to the photofragmentation average of 0.9 (indicated in Fig. 6), which further indicates that the reaction is complete on the ~1.5 ps time scale.

In $I_2^-(Ar)_9$, there is an increase in $\langle n_{I^-} \rangle$ of 1.5 between 2.8 and 12 ps, followed by a decrease of 0.5 through 100 ps. The increase in $\langle n_{I^-} \rangle$ does not occur for $I_2^-(Ar)_6$, and is most likely due to a transition to the $\tilde{X}$ or $\tilde{A}$ state, which is predicted to result in a substantially greater anion solvation, due to solvent rearrangement from the symmetric $\tilde{A}'$ state configuration to a more asymmetric configuration on the normal charge-switching state. This transition allows the neutral iodine atom to leave the cluster virtually unimpeded. A transition to the $\tilde{A}$ state without recombination is supported by Faeder *et al.*,[20] who found these transitions to be responsible for the high-mass $I^-(Ar)_n$ channel observed by Vorsa *et al.* in clusters of $I_2^-(Ar)_{n \geq 11}$,[6,7] since the solvent can more effectively surround the I^- in a normal charge-switching state. Although not reported in the paper, their model also observes transitions prior to final dissociation in clusters of $I_2^-(Ar)_9$.[37] In $I_2^-(CO_2)_n$ clusters, this electronic transition mechanism is present in all cluster sizes and occurs as rapidly as ~500 fs in large clusters.[35]

The long-time (> 12 ps) decrease in $\langle n_{I^-} \rangle$ is probably due to solvent evaporation. $\langle n_{I^-} \rangle$ exceeds the photofragmentation average (2.7) by 1.0 after the electronic transition

to the $\tilde{X}$ / $\tilde{A}$ state, and the value at 100 ps is still larger than the photofragmentation average by 0.5 Ar atoms. After solvent rearrangement on the normal charge-switching state, the marginal increase in available energy [estimated at 110 meV from the per atom evaporative energy loss of 73 meV] is likely to dissipate rather slowly through solvent evaporation, as observed for the I_2^- $\tilde{A}$ state in $I_2^-(Ar)_{12}$ and $I_2^-(Ar)_{16}$ (see below). The disagreement between the FPES and photofragmentation values of $\langle n_{I^-} \rangle$ at 100 ps probably indicates further evaporation on a longer time scale.

5.3. $I_2^-(Ar)_{12}$ and $I_2^-(Ar)_{16}$

In these two clusters, the increase in the eKE of the B features is similar to that of the smaller clusters, and also occurs over a ~1-2 ps timescale, as seen in Fig. 6. Thus, the increases, reflecting decreases in $\langle n_{I^-} \rangle$, are probably caused by the same mechanism of I^- pulling away from the cluster. However, in $I_2^-(Ar)_{16}$, the less-solvated I atom is expected to be more arrested by the surrounding Ar atoms than in the smaller clusters, so that the decrease in $\langle n_{I^-} \rangle$ may be partially attributed to evaporation of cluster atoms.

According to the photofragmentation study, there is substantial recombination of I_2^-, with virtually no I^- remaining in $I_2^-(Ar)_{16}$. Features E and F appear by ~10 ps in each cluster, indicating recombination of I_2^- on the $\tilde{X}$ state. The broadening of the D features after 2.0-2.5 ps, and the shifting toward lower eKE, particularly for $I_2^-(Ar)_{16}$ where the decrease (240 meV) is much larger than in $I_2^-(Ar)_9$, indicate recombination on the $\tilde{A}$ state. Because these spectra no longer indicated the presence of exclusively I^-, simulations were required in order to characterize the dynamics after these time delays.

In $I_2^-(Ar)_{12}$, although the D features broaden through the longest time delay measured, and the intensities of features E and F also grow slowly throughout this time

range, there is little change in shape to any of these features after their formation by ~10 ps. Therefore, only the spectrum at 45 ps is shown in Fig. 8(a), where feature E is most intense. Unfortunately, the poor signal-to-noise ratio in the region of this feature, even at 45 ps, made accurate determination of <v> difficult. We therefore set <v> = 68 and $\langle n_X \rangle$ = 0, the value consistent with the calculated available energy after evaporation of all 12 Ar atoms (710 meV) assuming 73 meV per Ar atom. Complete loss of solvent is consistent with Vorsa *et al.*,[6,7] who observed $\langle n_X \rangle$ = 0 in their photofragmentation study.

Using <v> as a starting point, an $\tilde{X}$ state vibrational distribution was constructed with P_X = 0.30 and $\langle n_X \rangle$ = 0. Although the $\tilde{X} \leftarrow \tilde{X}$ ITP transition does not accurately reproduce feature E, error bars in the intensity are estimated at 20-30%, and the overall intensity in this region is comparable to the simulation. The $\tilde{X} \leftarrow \tilde{X}$ OTP and $I_2^* \leftarrow \tilde{X}$ transitions together account for much of the broad feature F. P_X is close to the photofragmentation value (0.23). D_1 was represented by a combination of I_2^- $\tilde{A}$ and I^- states, using the following parameters: P_A = 0.21, P_{I^-} = 0.49, $\langle n_A \rangle$ = 5.0 and $\langle n_{I^-} \rangle$ = 3.2. The eKE of D_2 is also accounted for by the excited spin-orbit transitions from these states, though the intensity is lower than in the spectrum. The populations of the I_2^- $\tilde{A}$ and I^- channels are close to the photofragmentation values (0.23 and 0.54, respectively; see Table 5), and $\langle n_{I^-} \rangle$ is equal to the measured average. $\langle n_A \rangle$, however, is larger than the photofragmentation average of 2.3, which was necessary in order to adequately simulate the spectrum.

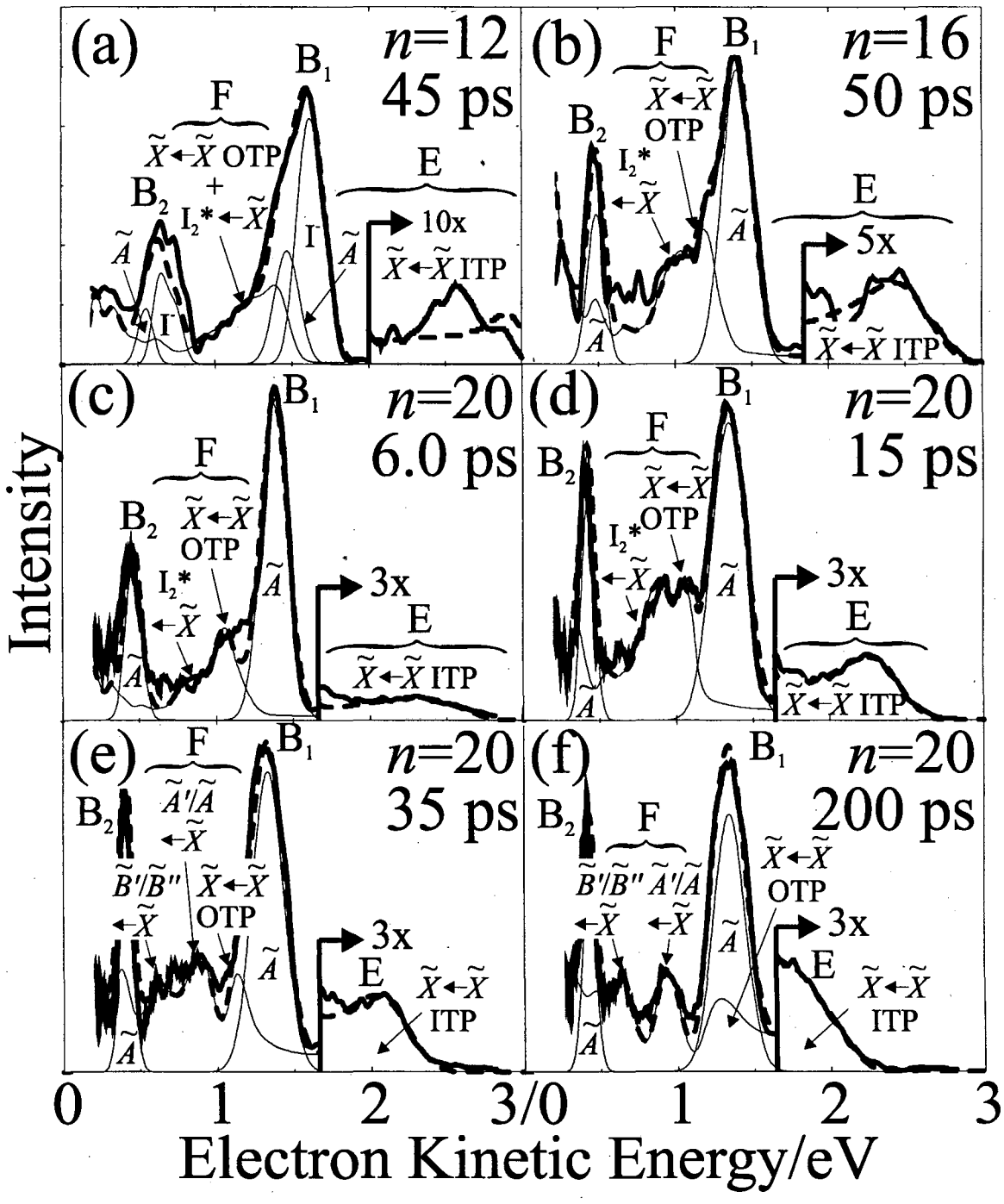


Fig. 8. Experimental (solid) and simulated (dotted) FPES of $I_2^-(Ar)_n$ clusters at selected time delays. Thin solid lines indicate simulated contributions from various states. (a) $I_2^-(Ar)_{12}$, 45 ps; (b) $I_2^-(Ar)_{16}$, 50 ps; (c) $I_2^-(Ar)_{20}$, 6.0 ps; (d) $I_2^-(Ar)_{20}$, 15 ps; (e) $I_2^-(Ar)_{20}$, 50 ps; (f) $I_2^-(Ar)_{20}$, 200 ps.

P_X is plotted *vs.* time in Fig. 9 for several time delays between 3.5 and 45 ps. Population appears by 9.0 ps, and grows in slowly through 45 ps. The slow, monotonic increase in the width of the D features over this time interval (not shown) indicates a similarly slow growth of the I_2^- $\tilde{A}$ state, and the large value of $\langle n_A \rangle$ at 45 ps compared to the photofragmentation average also indicates incomplete dynamics on this state. Both $\langle n_X \rangle$ and $\langle n_{I^-} \rangle$, however, are equal to the photofragmentation averages. In $I_2^-(Ar)_6$ and $I_2^-(Ar)_9$ clusters, $\langle n_{I^-} \rangle$ was also observed to be very close (≤ 0.5 atoms) to the Vorsa *et al.* results by ~50 ps. Thus, the I^- solvent evaporation appears to be complete by 45 ps, and only the $\tilde{A}$ state has not finished relaxing. This conclusion is supported by the Faeder *et al.* study, which reported that $\tilde{A}$ state evaporation required longer than the 50 ps length of the simulation to complete,[20,23] while the low-mass $I^-(Ar)_n$ product, which gives rise to the photoelectron signal at the highest eKE where the simulation is most sensitive to $\langle n_{I^-} \rangle$, appears in the first 1.0 ps. It would be helpful to measure $I_2^-(Ar)_{12}$ FPES spectra at longer time delays to verify that $\langle n_A \rangle$ decreases beyond 50 ps.

In the FPES of $I_2^-(Ar)_{16}$, the shifts in the D features are complete by 20 ps, and after features E and F have appeared at 10 ps, they evolve slightly through 50 ps, but the decrease in $\langle v \rangle$ on the $\tilde{X}$ state is small (5.1). Therefore, only the 50 ps spectrum is shown in Fig. 8(b), with the following parameters: $P_X = 0.50$, $P_A = 0.50$, $\langle v \rangle = 34.4$, $\langle n_X \rangle = 0$ and $\langle n_A \rangle = 8.0$. P_X and P_A are close to the photofragmentation results (0.43 and 0.57, respectively). $\langle v \rangle$ corresponds almost exactly to the calculated energy remaining in the cluster after evaporation of all 16 Ar atoms (420 meV). This result supports the assumption that $\langle n_X \rangle = 0$, which is also the long-time limit set by the photofragmentation study. Feature E is reproduced by the $\tilde{X} \leftarrow \tilde{X}$ ITP transition, while F is accounted for

by the overlapping $\tilde{X} \leftarrow \tilde{X}$ OTP and $I_2^* \leftarrow \tilde{X}$ transitions. Features D_1 and D_2 were simulated by the $\tilde{A}$ state. The value of $\langle n_A \rangle$ (8.0) is actually larger than $\langle n_I \rangle$ at 2.0 ps (5.0), reflecting the expected solvent rearrangement on the normal charge-switching state, much as was seen in $I_2^-(Ar)_9$. However, as $\langle n_A \rangle$ is larger than the photofragment average (6.2), further evaporation must occur on a longer time scale, as also inferred for $I_2^-(Ar)_{12}$.

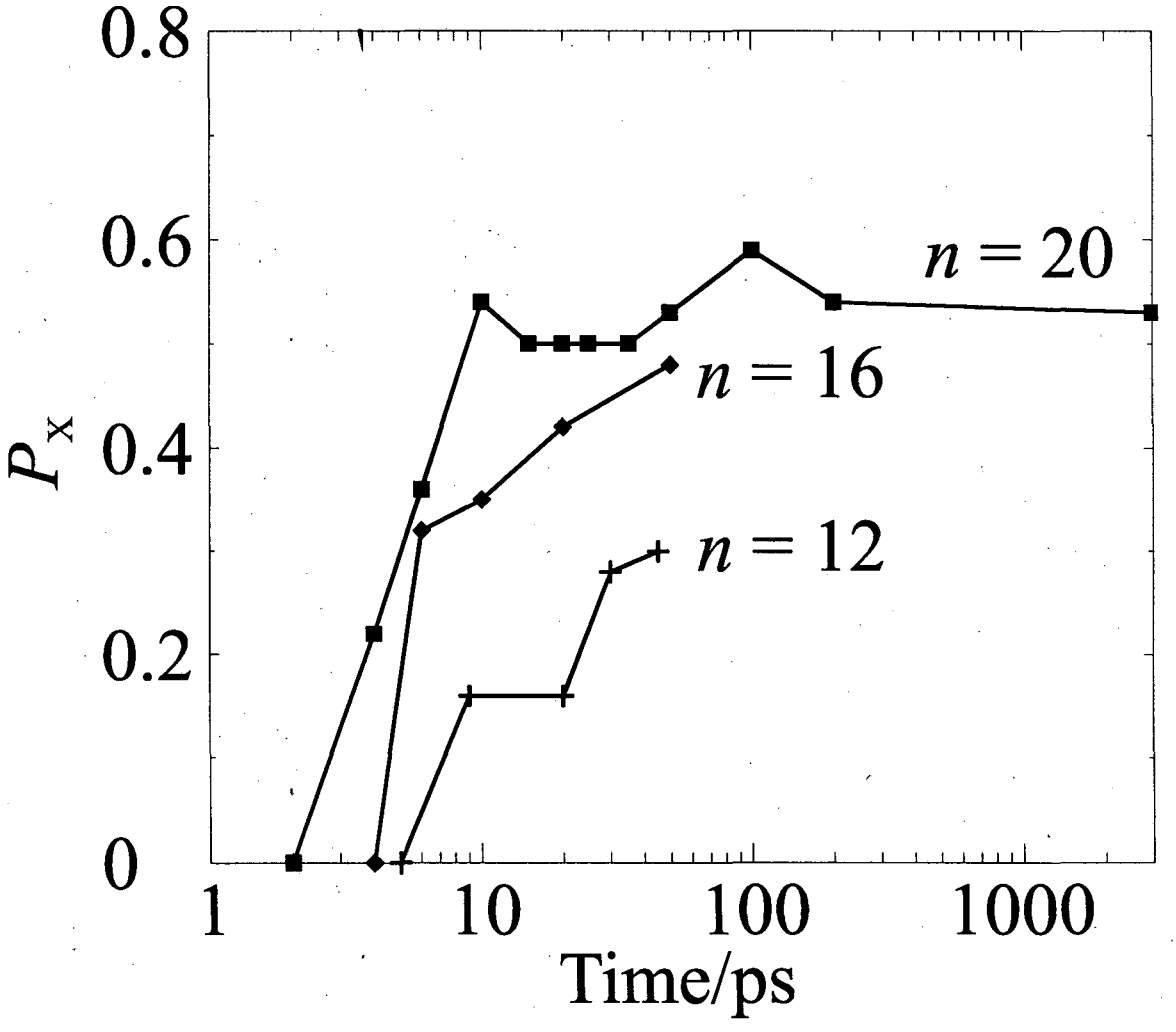


Fig. 9. Population of I_2^- $\tilde{X}$ state (P_X) *vs.* time, for $I_2^-(Ar)_{n \geq 12}$ clusters: $I_2^-(Ar)_{12}$ (plus-signs), $I_2^-(Ar)_{16}$ (diamonds), $I_2^-(Ar)_{20}$ (squares). Detached points indicate photofragmentation averages from Vorsa *et al.*[6]

P_X is plotted *vs.* time in Fig. 9 for several simulated time delays. It rises faster, and is larger at all time delays than that of $I_2^-(Ar)_{12}$, though it never achieves a plateau

value. As P_X for time delays ≥ 20 ps are roughly the same as the photofragmentation value, however, population transfer to the $\tilde{X}$ state ("recombination") is probably complete on this time scale (~20-50 ps). It appears that vibrational relaxation is also complete on this time scale, as <v> agrees with the long-term value at 50 ps.

5.4. $I_2^-(Ar)_{20}$

In this cluster, the decrease in $\langle n_I^- \rangle$ of the B features through 1.0 ps is smaller than in the smaller clusters (2.6 *vs.* ~3-4), but its time evolution is fairly similar. Since both I atoms are predicted to be completely surrounded by solvent atoms,[20] the unimpeded motion of I^- away from the Ar atoms in the smaller clusters cannot explain the observations. Instead, a combination of cluster expansion and rapid evaporation of Ar atoms, particularly the "capping" atoms which lie along the I_2^- axis, appears reasonable.[20] At later time delays, the features are relabeled D, where they broaden considerably, and shift 140 meV toward lower eKE; again, the I_2^- $\tilde{A}$ state is implicated. The time-resolved motion of the E feature, and the complex evolution of the F feature, also indicate extensive vibrational relaxation on the I_2^- $\tilde{X}$ state. Several time delays were simulated (6.0 ps, 15 ps, 50 ps, 200 ps) to follow this process, shown in Figs. 8(c-f).

In the 6.0 ps spectrum (P_X = 0.36, P_A = 0.64, <v> = 44.5, $\langle n_X \rangle$ = 8.0, $\langle n_A \rangle$ = 9.0), the I_2^- $\tilde{X}$ state is still in the process of growing in. It is also so vibrationally excited that the relative intensity of the $\tilde{X} \leftarrow \tilde{X}$ OTP transition appears to be changing rapidly in this range; thus, feature F above 1.0 eV is difficult to simulate accurately. At lower eKE, however, the $I_2^* \leftarrow \tilde{X}$ transitions accounts for much of the intensity of feature F, while the $\tilde{X} \leftarrow \tilde{X}$ ITP transition reproduces feature E satisfactorily. $\langle n_X \rangle$ is also

difficult to determine from feature F, but an approximate upper limit is obtained by matching the falling edge near 1.0 eV. The $\tilde{A}$ state, accounting for D_1 and D_2, has a smaller apparent number of Ar atoms than at later time delays, and D_1 is also not as broad. This probably reflects a wavepacket that is still in the process of moving into the $\tilde{A}$ state well, giving an artificially low value of $\langle n_A \rangle$.

The 15 ps spectrum ($P_X = 0.50$, $P_A = 0.50$, $\langle v \rangle = 29.0$, $\langle n_X \rangle = 6.0$, $\langle n_A \rangle = 11.0$) displays an $\tilde{X}$ state which is considerably larger in population, less vibrationally excited, and with fewer Ar atoms than at 6.0 ps. Feature E is well-reproduced by the $\tilde{X} \leftarrow \tilde{X}$ ITP transition. The F feature displays two peaks which are reproduced in the simulation, corresponding to the $\tilde{X} \leftarrow \tilde{X}$ OTP transition at higher eKE, and the $I_2^* \leftarrow \tilde{X}$ transitions at lower eKE, which enabled accurate determination of $\langle n_X \rangle$. The $\tilde{A}$ state, accounting for D_1 and D_2, has reached its asymptotic number of Ar atoms, although some variation (±0.5) is seen at other time delays. D_1 undergoes no more broadening at later time delays.

At 35 ps ($P_X = 0.50$, $P_A = 0.50$, $\langle v \rangle = 14.2$, $\langle n_X \rangle = 3.0$, $\langle n_A \rangle = 11.5$), the $\tilde{X}$ state has undergone more vibrational relaxation, so that the $\tilde{A}' / \tilde{A} \leftarrow \tilde{X}$ and $\tilde{B}' / \tilde{B}'' \leftarrow \tilde{X}$ transitions now appear as separate peaks at 900 and 610 meV, respectively. The $\tilde{X} \leftarrow \tilde{X}$ OTP transition appears as a small shoulder on the low eKE side of feature D_1; at later time delays, it is completely absorbed by this feature. The $\tilde{A}$ state, accounting for D_1 and D_2, is virtually unchanged from 15 ps.

The 200 ps spectrum ($P_X = 0.54$, $P_A = 0.56$, $\langle v \rangle = 5.6$, $\langle n_X \rangle = 0.5$, $\langle n_A \rangle = 11.0$) shows that I_2^- $\tilde{X}$ is almost completely relaxed. While the $\tilde{X} \leftarrow \tilde{X}$ ITP transition

accounts for feature E, the OTP transition falls completely under D_1. Feature F consists of the $\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$ and $\tilde{B}'/\tilde{B}'' \leftarrow \tilde{X}$ transitions, which are clearly separated with considerably less intensity between the peaks than at 35 ps, allowing accurate determination of $\langle n_X \rangle$. The $\tilde{A}$ state again accounts for D_1 and D_2.

P_X is plotted *vs.* time in Fig. 9, where it is seen to rise more rapidly than in the smaller clusters, achieving a plateau near ~0.55 by 10 ps, before either vibrational relaxation or solvent evaporation has neared completion. Thus, recombination appears to occur rapidly, after which these relaxation processes take place. This distinction between recombination and relaxation was not visible in $I_2^-(Ar)_{12}$ or $I_2^-(Ar)_{16}$, where relaxation has proceeded almost as far as possible by the time significant $\tilde{X}$ population was present. This trend of accelerating recombination with cluster size reflects the increased translational energy dissipation by a larger number of Ar atoms.

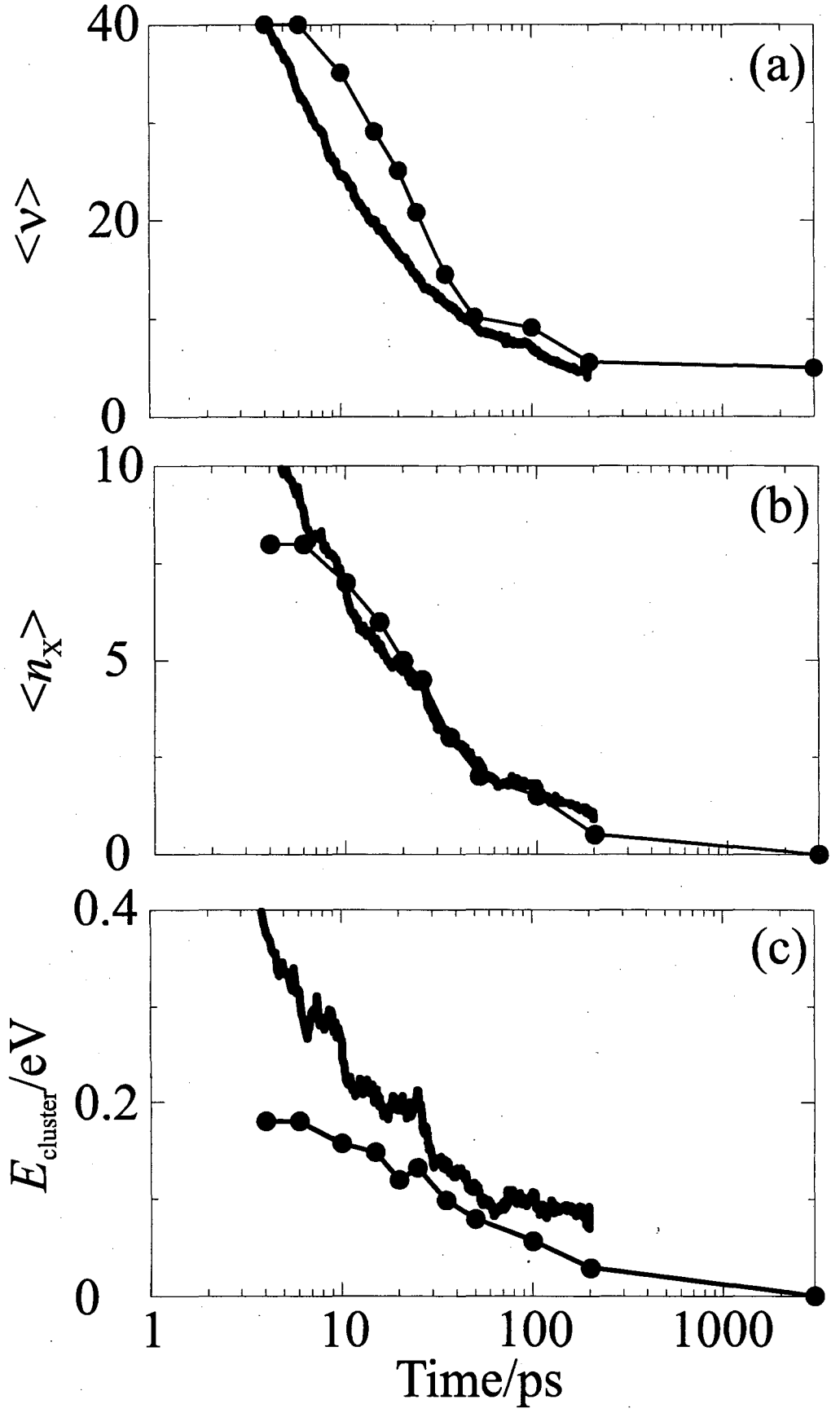
$\tilde{A}$ state relaxation appears complete by ~15 ps, soon after P_X reaches its final value at 10 ps, as indicated by the plateauing of $\langle n_A \rangle$ in the simulations. One should not be surprised by the agreement between $\langle n_A \rangle$ at long time delays (11.0±0.5) and the photofragmentation average (11.1), as the $\tilde{A}$ state parameters were determined by setting $\langle n_A \rangle$ in the 3 ns spectrum equal to the photofragmentation value (see Analysis section). It is interesting, however, that $\langle n_A \rangle$ is significantly larger (~2-3) at ~50 ps than the photofragmentation average in the $I_2^-(Ar)_{12}$ and $I_2^-(Ar)_{16}$ clusters. This trend is consistent with the increasing $\tilde{A}$ state recombination rate as cluster size increases. Faeder *et al.* predict slow (> 50 ps) evaporation of solvent from the I_2^- $\tilde{A}$ state, but they report a similarly slow rate for $I_2^-(Ar)_{20}$ as well.[20,23] As the $\tilde{A}$ state potential determined from the $I_2^-(Ar)_{20}$ spectrum differs considerably from the *ab initio* potential,[18] it is possible that the

presence of ~11 Ar atoms has a sizable influence on this marginally-bound state, so that potential parameters will be different in smaller clusters. Spectra of $I_2^-(Ar)_{12}$ and $I_2^-(Ar)_{16}$ at longer time delays are necessary to resolve whether slow evaporation, or changes in the I_2^- $\tilde{A}$ state well depth and/or equilibrium distance, is responsible for the discrepancy in $\langle n_A \rangle$ for $I_2^-(Ar)_{12}$ and $I_2^-(Ar)_{16}$ clusters.

The average vibrational level <v> of the $\tilde{X}$ state simulated for each measured spectrum has been plotted *vs.* time in Fig. 10(a), along with <v> derived from Faeder *et al.*[23] Between 6.0 ps and 3 ns, the FPES <v> drops from 40.0 to 5.1, with most of the decrease occurring before 50 ps. There is little difference in <v> between 200 ps and 3 ns. The Faeder *et al.* results have been adjusted to reflect the available energy from a 780 nm photon, rather than 790 nm as used in the study. The time scale for vibrational relaxation in the Faeder *et al.* results is similar, and while there is a sizable discrepancy between 6.0 and 35 ps, the curves match fairly well at later time delays.

The average number of solvent atoms $\langle n_X \rangle$ is plotted *vs.* time in Fig. 10(b), together with the Faeder *et al.* simulation results. In the FPES data, $\langle n_X \rangle$ drops from 8.0 to 0.0 between 6.0 ps and 3 ns, with the majority of the Ar loss (6.0) occurring before 50 ps. The final ~2.0 Ar atoms take longer than 150 ps to evaporate, consistent with the slowdown in evaporation rates seen in other clusters. The Faeder *et al.* data display remarkable agreement with the experiment at time delays ≥ 6 ps. The agreement is encouraging, for it not only suggests that the model is correctly describing the mechanism

Fig. 10. (On next page) (a) Average I_2^- $\tilde{X}$ state vibrational level (<v>), (b) average number of I_2^- $\tilde{X}$ state solvent atoms ($\langle n_X \rangle$), and (c) excess cluster energy ($E_{cluster}$) *vs.* time, for $I_2^-(Ar)_{20}$: From FPES (circles and thin line); from Faeder *et al.*[23] (thick line).


40
20
0
<v>
(a)
10
5
0
<n_x>
(b)
0.4
0.2
0
$E_{cluster}$/eV
(c)
1
10
100
1000
Time/ps

of vibrational relaxation and solvent evaporation, but it confirms that the spectroscopic method used to determine $\langle n_X \rangle$ is valid.

At first glance, it appears that vibrational relaxation and solvent evaporation evolve in step with one another, but closer examination reveals a significant time lag. To make this comparison, E_{solv}, the total solvent energy that can be removed by evaporation of Ar atoms, is defined:

$$E_{solv} = \langle n_X \rangle \Delta E_{evap} \tag{2}$$

where ΔE_{evap} is the average energy lost from the cluster by evaporation of one Ar atom (73 meV).[7] E_{int}, the average I_2^- internal energy in excess of the final (3 ns FPES) energy, is obtained by:

$$E_{int} = E(\langle v \rangle) - E(\langle v \rangle_f) \tag{3}$$

where $E(\langle v \rangle)$ is the Morse energy for vibrational level $\langle v \rangle$, and $\langle v \rangle_f$ (= 5.1) is the average vibrational level at 3 ns.

Subtracting E_{int} from E_{solv}, one obtains a positive excess "cluster" energy $E_{cluster}$, indicating energy neither associated with the I_2^- vibrational mode, nor lost as evaporating solvent. $E_{cluster}$ is plotted *vs.* time in Fig. 10(c), along with the same quantity calculated using the data from Faeder *et al.*[23] (adjusted for 780 nm, and assuming $\langle v \rangle_f$ is equal to the FPES value). Both plots show a general decrease with time, the experimental $E_{cluster}$ dropping from 180 meV at 6.0 ps, to 0 by 3 ns. The model shows higher values at all time delays, but it is more pronounced at earlier times (320 meV at 6.0 ps), the disparity being due to the smaller $\langle v \rangle$ relative to the experimental data at these time delays.

The excess cluster energy (at 6.0 ps, equivalent to 2.5 extra Ar atoms in the experimental data, and 4.4 atoms in the simulation) implies that energy is temporarily

tied up in solvent modes after removal from the I_2^- vibrational coordinate, but before solvent evaporation. The amount of excess energy is expected to be larger at early times, because there are more solvent atoms available to provide for storage of this energy. For solvent molecules with a stronger binding energy to I_2^- (such as CO_2), this storage capacity is larger, allowing greater amounts of energy to be stored for longer times.[35] Although there is disagreement between the Faeder *et al.* model and the FPES data in <v>, which in turn affects $E_{cluster}$, the signature of a delayed evaporation mechanism is undeniable in both model and experiment, and in fact, the model results show a more pronounced effect.

6. Conclusions

FPES has been used to study the photodissociation, recombination and energy transfer dynamics of $I_2^-(Ar)_n$ clusters over a range of sizes. From determination of the number of Ar atoms surrounding the nascent I^- product, the anomalous charge-switching nature of the $\tilde{A}'$ state is confirmed, with the electron localized on the less-solvated I atom immediately after photoexcitation. Subsequent separation of I^- and I fragments results in a decreasing number of Ar atoms through ~1.5 ps in all clusters, after which the dissociated products have been formed in the case of small (n = 6) clusters, or recombination on the I_2^- $\tilde{X}$ or $\tilde{A}$ states begins to occur in larger ($n \geq 12$) clusters. Although only dissociated products are observed for $I_2^-(Ar)_9$, there is a long-time (~15-50 ps) increase in the number of Ar atoms in the cluster, suggesting an electronic transition to the $\tilde{X}$ or $\tilde{A}$ state followed by solvent rearrangement. This effect was predicted by Faeder *et al.*,[20,37] and illustrates a situation intermediate between dissociation and caging, whereby an electronic transition occurs without subsequent recombination.

In $I_2^-(Ar)_{12}$ and $I_2^-(Ar)_{16}$, vibrational relaxation on the $\tilde{X}$ state was slight or unobservable, and the final vibrational level remained quite high (<v> = 68 and 34, respectively). In $I_2^-(Ar)_{20}$, however, extensive vibrational relaxation was observed, accompanied by evaporation of solvent. Maximum relaxation (to <v> = 5.1) is achieved by 3 ns, with the loss of all Ar atoms. The average vibrational level <v> and number of solvent atoms $\langle n_X \rangle$ were compared to the theoretical study of Faeder *et al.*,[23] which agreed in large measure, despite a discrepancy in <v> between 6.0 and 35 ps. Further analysis revealed excess energy stored in the cluster, demonstrating a delay between removal of I_2^- vibrational energy to the cluster, and its dissipation through solvent evaporation, in both the experimental and theoretical studies.

Although I_2^- $\tilde{X}$ state solvent evaporation is complete on the time scale of the experiment, I^- and I_2^- $\tilde{A}$ state cluster evaporation appears to require longer to complete in $I_2^-(Ar)_{n \leq 16}$. In $I_2^-(Ar)_9$, there is a long-time slope in the graph of $\langle n_{I^-} \rangle$ *vs.* time (Fig. 6), and the final $\langle n_{I^-} \rangle$ is slightly (0.5 atoms) higher than the photofragmentation results.[6,7] Larger discrepancies exist for the I_2^- $\tilde{A}$ state in $I_2^-(Ar)_{12}$ and $I_2^-(Ar)_{16}$, where the long-time value of $\langle n_A \rangle$ is higher by ~2-3 atoms, but in $I_2^-(Ar)_{20}$, $\langle n_A \rangle$ matches the photofragmentation value. The differences either indicate much slower solvent evaporation in the smaller clusters, as predicted by theoretical studies,[20,23] or a change in the $\tilde{A}$ state parameters from $I_2^-(Ar)_{12}$ through $I_2^-(Ar)_{20}$, which would preclude comparative analysis. FPES experiments on $I_2^-(Ar)_{12}$ and $I_2^-(Ar)_{16}$ at longer time delays would help resolve this ambiguity.

7. Acknowledgments

The authors would like to thank James Faeder, Nicole Delaney and Professor Robert Parson for many helpful discussions. Victor Batista is also acknowledged for sharing his insight. Funds were supplied by the National Science Foundation under Grant No. CHE-9710243, and the Defense University Research Instrumentation Program under Grant No. F49620-95-1-0078, and are gratefully acknowledged.

8. References

1 M. L. Alexander, N. E. Levinger, M. A. Johnson, D. Ray, and W. C. Lineberger, J. Chem. Phys. **88,** 6200 (1988).

2 J. M. Papanikolas, J. R. Gord, N. E. Levinger, D. Ray, V. Vorsa, and W. C. Lineberger, J. Phys. Chem. **95,** 8028 (1991).

3 M. E. Nadal, P. D. Kleiber, and W. C. Lineberger, J. Chem. Phys. **105,** 504 (1996).

4 S. Nandi, A. Sanov, N. Delaney, J. Faeder, R. Parson, and W. C. Lineberger, J. Phys. Chem. A **102,** 8827-8835 (1998).

5 A. Sanov, S. Nandi, and W. C. Lineberger, J. Chem. Phys. **108,** 5155 (1998).

6 V. Vorsa, Ph.D. Thesis, University of Colorado, Boulder (1996).

7 V. Vorsa, P. J. Campagnola, S. Nandi, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **105,** 2298 (1996).

8 D. Ray, N. E. Levinger, J. M. Papanikolas, and W. C. Lineberger, J. Chem. Phys. **91,** 6533 (1989).

9 J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, J. R. Gord, and W. C. Lineberger, J. Chem. Phys. **97,** 7002 (1992).

[10] J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, H. K. Buchenau, and W. C. Lineberger, J. Chem. Phys. **99,** 8733 (1993).

[11] V. Vorsa, S. Nandi, P. J. Campagnola, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **106,** 1402 (1997).

[12] B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Science **276,** 1675 (1997).

[13] B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Faraday Discuss. **108,** 101 (1997).

[14] L. Perera and F. G. Amar, J. Chem. Phys. **90,** 7354 (1989).

[15] F. G. Amar and L. Perera, Z. Phys. D. **20,** 173-175 (1991).

[16] P. E. Maslen, J. M. Papanikolas, J. Faeder, R. Parson, and S. V. ONeil, J. Chem. Phys. **101,** 5731 (1994).

[17] J. M. Papanikolas, P. E. Maslen, and R. Parson, J. Chem. Phys. **102,** 2452 (1995).

[18] P. E. Maslen, J. Faeder, and R. Parson, Chem. Phys. Lett. **263,** 63 (1996).

[19] V. S. Batista and D. F. Coker, J. Chem. Phys. **106,** 7102 (1997).

[20] J. Faeder, N. Delaney, P. E. Maslen, and R. Parson, Chem. Phys. Lett. **270,** 196 (1997).

[21] N. Delaney, J. Faeder, P. E. Maslen, and R. Parson, J. Phys. Chem. A **101** (1997).

[22] B. M. Ladanyi and R. Parson, J. Chem. Phys. **107,** 9326 (1997).

[23] J. Faeder and R. Parson, J. Chem. Phys. **108,** 3909 (1998).

[24] P. E. Maslen, J. Faeder, and R. Parson, Mol. Phys. **94,** 693 (1998).

[25] J. Faeder, N. Delaney, P. E. Maslen, and R. Parson, Chem. Phys. **239,** 525 (1998).

[26] M. E. Nadal, S. Nandi, D. W. Boo, and W. C. Lineberger, J. Chem. Phys., submitted .

[27] C. J. Margulis and D. F. Coker, J. Chem. Phys., in press (1999).

28 A. E. Johnson, N. E. Levinger, and P. F. Barbara, J. Phys. Chem. **96,** 7841 (1992).

29 D. A. V. Kliner, J. C. Alfano, and P. F. Barbara, J. Chem. Phys. **98,** 5375 (1993).

30 J. C. Alfano, Y. Kimura, P. K. Walhout, and P. F. Barbara, Chem. Phys. **175,** 147-155 (1993).

31 I. Benjamin, P. F. Barbara, B. J. Gertner, and J. T. Hynes, J. Phys. Chem. **99,** 7557-7567 (1995).

32 P. K. Walhout, J. C. Alfano, K. A. M. Thakur, and P. F. Barbara, J. Phys. Chem. **99,** 7568 (1995).

33 K. Asmis, T. Taylor, and D. M. Neumark, J. Chem. Phys. **109,** 4389 (1998).

34 M. T. Zanni, T. R. Taylor, B. J. Greenblatt, B. Soep, and D. M. Neumark, J. Chem. Phys. **107,** 7613 (1997).

35 B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, J. Chem. Phys., to be submitted .

36 B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Chem. Phys. Lett. **258,** 523 (1996).

37 N. Delaney, J. Faeder, and R. Parson, private communication .

38 W. C. Wiley and I. H. McLaren, Rev. Sci. Instrum. **26,** 1150 (1955).

39 O. Cheshnovsky, S. H. Yang, C. L. Pettiette, M. J. Craycraft, and R. E. Smalley, Rev. Sci. Instrum. **58,** 2131 (1987).

40 L.-S. Wang, H.-S. Cheng, and J. Fan, J. Chem. Phys. **102,** 9480 (1995).

41 M. T. Zanni, V. S. Batista, B. J. Greenblatt, W. H. Miller, and D. M. Neumark, J. Chem. Phys. **110,** 3748 (1999).

42 C. E. Moore, *Atomic Energy Levels, Vol. 1, NSRDS-NBS 35* (1971).

[43] I. Yourshaw, Y. Zhao, and D. M. Neumark, J. Chem. Phys. **105,** 351 (1996).

[44] T. Kuhne and P. Vohringer, J. Chem. Phys. **105,** 10788 (1996).

[45] S. E. Bradforth, Ph.D. Thesis, University of California, Berkeley (1992).

Chapter 7. Femtosecond photoelectron spectroscopy of $I_2^-(CO_2)_n$ photodissociation dynamics (n = 4, 6, 9, 12, 14, 16)*

The photodissociation dynamics of I_2^- embedded in size-selected van der Waals clusters of 4-16 CO_2 molecules have been studied with femtosecond photoelectron spectroscopy (FPES). The range of cluster sizes span the uncaged and fully-caged product limits for this reaction; several intermediate-sized clusters exhibit both caged and uncaged products at long (200 ps) time delay. All clusters exhibit exclusively solvated I^- features initially, with the number of CO_2 molecules increasing with time delay, due to solvent rearrangement and/or electronic relaxation in which the electron hops from the less- to more-solvated I atom. At longer time delays (~500 fs-10 ps), vibrationally excited I_2^- $\tilde{X}$ state features appear in $I_2^-(CO_2)_{n \geq 6}$ clusters, with a decrease in I^- intensity. The $\tilde{X}$ state features evolve through 10-200 ps, reflecting vibrational relaxation, with a rate which increases substantially with cluster size. In clusters of $I_2^-(CO_2)_{n \geq 9}$, there is also significant intensity (~0.1-0.4 fraction of the total population) arising from a solvent-separated I_2^- structure, which resembles I^- with fewer surrounding CO_2 molecules, presumably trapped on the $\tilde{A}$ state. Detailed simulations of the spectra were performed in order to determine the populations, $\tilde{X}$ state vibrational level, and number of CO_2 molecules in the cluster at different time delays. Results are compared to previous experimental and theoretical studies of $I_2^-(CO_2)_n$ photodissociation, as well as to the FPES study of $I_2^-(Ar)_n$ photodissociation presented elsewhere in this journal.

---

* B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, J. Chem. Phys., in preparation for submission.

Chapter 7

1. Introduction

The gas-phase study of I_2^- embedded in a van der Waals cluster of CO_2 molecules is a powerful method for understanding the effects of solvation on a chemical reaction. Solvent molecules, which often profoundly change the energetic surfaces governing a reaction, may be progressively added to a cluster one molecule at a time, owing to the inherent size-selectivity of charged particles. This approach enables changes in reaction dynamics to be studied across cluster size, as the solvent environment moves from the gas phase to a condensed-like phase. Lineberger and coworkers first examined the anionic products of $Br_2^-(CO_2)_n$ photodissociation in 1988,[1] and subsequently expanded their scope to include I_2^- in CO_2, Ar and most recently OCS clusters, examining both asymptotic products as well as time-resolved dynamics.[2-10] Theoretical investigations of the structure and dynamics of I_2^- clusters have been performed by several groups.[11-23] Solution-phase experiments involving I_2^- photodissociation in numerous solvents have also been pursed by Barbara and coworkers.[24-28] The gas-phase studies have shown that the dissociation channel closes in I_2^- clusters with a relatively small number of solvent molecules (less than one-half of a solvent shell), and that the rate of appearance of "caged" I_2^- products increases dramatically with cluster size. Comparisons across solvent type reveal a strong dependence on solvent-solute binding energy in the recombination rate. In solution, timescales for caging are similar to those of large $I_2^-(CO_2)_n$ clusters, though interesting new effects appear, such as the permanent escape of I^- into solution, not observed in the gas-phase experiments.

Our research group has focused on $I_2^-(Ar)_n$ and $I_2^-(CO_2)_n$ clusters using femtosecond photoelectron spectroscopy (FPES).[29,30] The preceding article on $I_2^-(Ar)_n$

photodissociation dynamics[31] demonstrated the effect of a weakly-interacting solvent on the I_2^- photodissociation reaction, such as closing of the dissociation channel through solvent-induced electronic transitions, dissipation of energy through solvent evaporation, and stabilization of an excited state. It was also shown that the FPES pump-probe technique[32] provided an enormous amount of time-resolved information, including the changing electronic and vibrational state of the anion, and the number of solvent molecules near the negative charge. These effects were not observable using previous experimental approaches.

The $I_2^-(CO_2)_n$ system represents a stronger solute-solvent interaction than that of $I_2^-(Ar)_n$. While CO_2 lacks a dipole moment, its large quadrupole moment gives rise to a sizable I_2^--CO_2 well depth (~200 meV),[7] about four times larger than that of I_2^--Ar (53 meV).[33] Solvent-induced effects are therefore expected to be more pronounced in $I_2^-(CO_2)_n$ clusters. This paper reports on the use of FPES to investigate $I_2^-(CO_2)_n$ clusters, ranging in size from $n = 4$, which produces almost exclusively uncaged $I^-(CO_2)_n$ fragments, to $n = 16$, which exhibits only caged $I_2^-(CO_2)_n$ fragments. Intermediate-sized clusters are of particular interest, as both types of fragments are present. The timescales for caging, vibrational relaxation and solvent rearrangement and evaporation are investigated as a function of cluster size. A comparison with the $I_2^-(Ar)_n$ study is made at the end of the paper.

The FPES technique excites a cold, mass-selected $I_2^-(CO_2)_n$ cluster with a 780 nm femtosecond pump pulse from the $\tilde{X}(^2\Sigma_u^+)$ ground state to the $\tilde{A}'(^2\Pi_{1/2,g})$ dissociative state of I_2^- (Fig. 1). A second, delayed femtosecond probe pulse (260 nm) then detaches an electron from the cluster, producing a photoelectron spectrum. Different vibrational

and electronic states of the anion, including photodissociated I, are readily distinguishable, and the number of solvent molecules nearby to the anion can also be determined, giving a detailed picture of the dynamics. The information obtained with FPES is complementary to previous experimental and theoretical work, which is described briefly below.

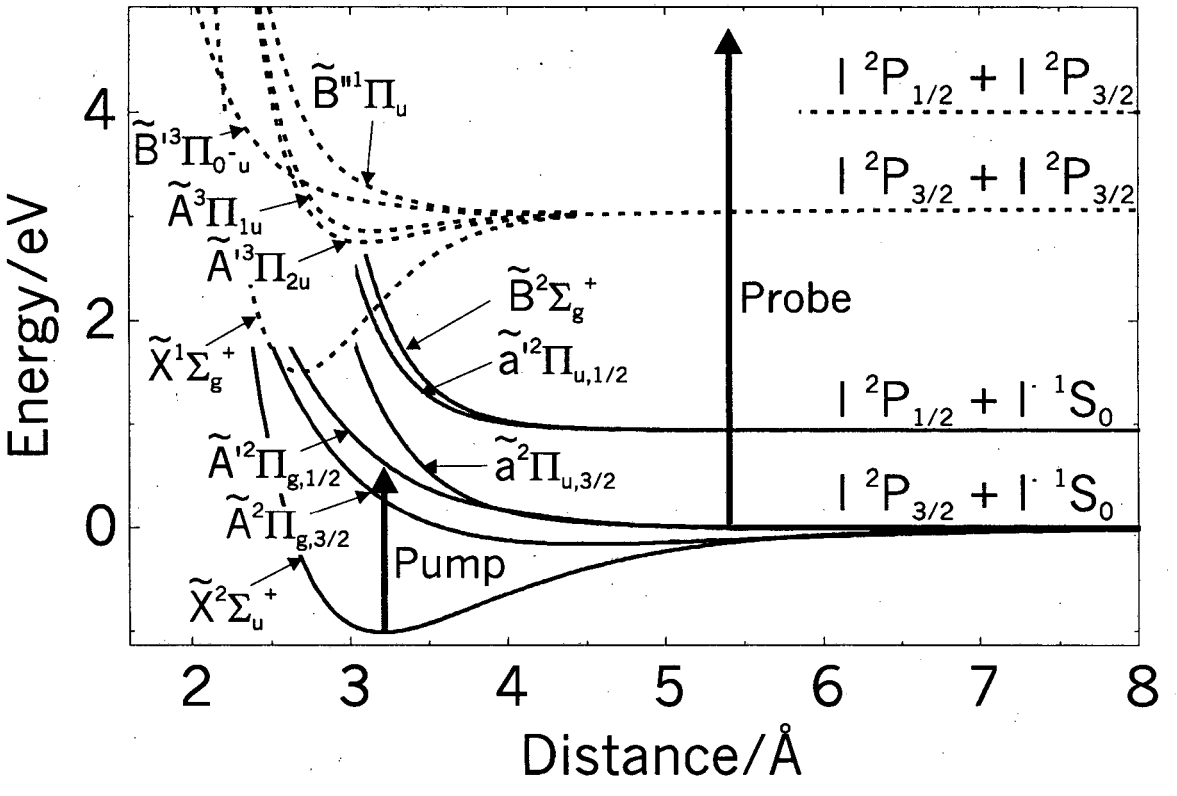


Fig. 1. Potential energy curves for bare I_2^-. Solid lines (--): I_2^-. Dotted lines (·····): I_2.

Photofragment mass spectra of photodissociated $I_2^-(CO_2)_n$ clusters were measured by the Lineberger group at two wavelengths, 720 and 790 nm.[4,7] In these experiments, cold mass-selected clusters were excited with a pulsed laser, and photofragment masses were determined using a reflectron. Only uncaged fragments were observed in smaller clusters, gradually replaced by caged fragments as the size increased; the uncaged channel closed at one full solvent shell, $I_2^-(CO_2)_{16}$. The number of solvent molecules

present in both types of product fragments was smaller than that of the parent cluster, with more molecules lost in the caged fragments due to energy released when the I_2^- bond reforms.

Time-resolved absorption recovery experiments on $I_2^-(CO_2)_n$ clusters were also performed by the Lineberger group, in which cold mass-selected clusters were excited with a ps-duration laser pulse at 720 nm,[6-9] or a fs-duration pulse at 790 nm,[10] then re-excited with a second, identical pulse after a variable time delay. The total flux of two-photon photofragments as a function of pump-probe time delay was assumed to indicate the absorption of I_2^- near the bottom of the $\tilde{X}$ state well. Minimum absorption was observed initially, growing at later time delays. Exponential recovery time constants ($t_{1/e}$ = 1.3-24 ps) were much shorter than observed for $I_2^-(Ar)_{20}$ (127 ps).[10] Additional temporal structure in the recovery curves was also observed, most notably a transient increase ("bump") at ~2 ps in $I_2^-(CO_2)_{14\text{-}16}$ clusters, attributed to a resonant $\tilde{a}(^2\Pi_{3/2,u}) \leftarrow \tilde{A}(^2\Pi_{3/2,g})$ transition in I_2^-.

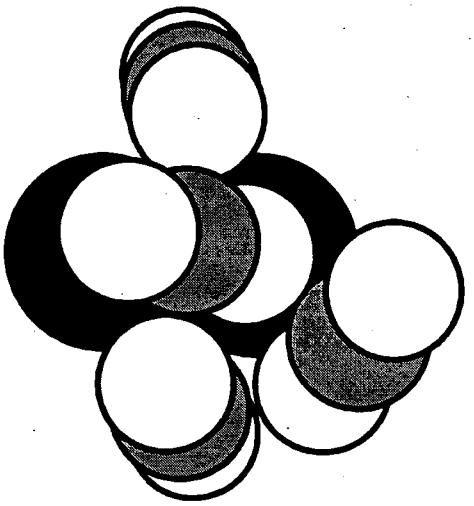
Considerable theoretical work has been done on the $I_2^-(CO_2)_n$ system. Minimum energy structures have been investigated by several groups, and agree in large measure.[12,14,18,23,34] Fig. 2 shows calculated structures for selected cluster sizes, taken from the Parson group studies.[14,18,34] The first 3 solvent molecules locate in a ring about the I_2^- internuclear axis. Six additional molecules surround one I atom, until no more space remains. The other end of the I_2^- molecule is then solvated, and the number of solvent molecules around the I_2^- internuclear axis also increases to 4. A full shell is obtained for 16 solvent molecules. The charge is equally shared by both I atoms in the small and large

clusters, but in the $n \approx 6$-12 size regime, where asymmetric structures are observed, the charge is more localized on the heavily solvated atom.

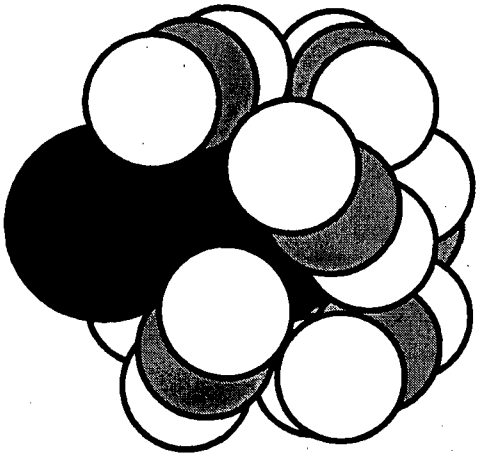
The mechanism of $I_2^-(CO_2)_n$ photodissociation was investigated by Parson and coworkers[13,14,18,19,22,34] using nonadiabatic molecular dynamics simulations. They found that photodissociation on the $\tilde{A}'$ state is accompanied by rearrangement of solvent to a more symmetrical configuration in the first 400-500 fs in all clusters, due to an effect called "anomalous charge-switching" where the solvent is prevented from fully surrounding the charge. A transition to the $\tilde{A}$ or $\tilde{X}$ state occurs in all clusters, even those which form uncaged products, from ~500 fs to > 2 ps; the longest transition times occur for intermediate-sized clusters [$I_2^-(CO_2)_{n\approx 9}$]. Although some transitions proceed from the $\tilde{A}'$ directly to the $\tilde{X}$ state, the $\tilde{A} \leftarrow \tilde{A}'$ transition becomes increasingly favored as an intermediate step as cluster size increases, with the effect that $\tilde{X}$ state recombination becomes more and more delayed in large clusters. However, once recombination on the $\tilde{X}$ state occurs, vibrational relaxation proceeds rapidly (1-3 ps) in all clusters. Population on the $\tilde{A}$ state is not permanent, but persists in $I_2^-(CO_2)_{n\geq 7}$ anywhere from 1 to 100 ps. Unlike the $\tilde{A}$ state product predicted for $I_2^-(Ar)_n$ clusters, with I_2^- stabilized in the shallow well, this structure consists of a "solvent-separated pair," with the I^- surrounded by most of the solvent molecules, and the I located outside the solvent shell, surrounded by a fewer number of CO_2 molecules.

Fig. 2. (On next page) Calculated minimum-energy structures of selected $I_2^-(CO_2)_n$ clusters: (a) $I_2^-(CO_2)_4$; (b) $I_2^-(CO_2)_9$; (c) $I_2^-(CO_2)_{16}$.

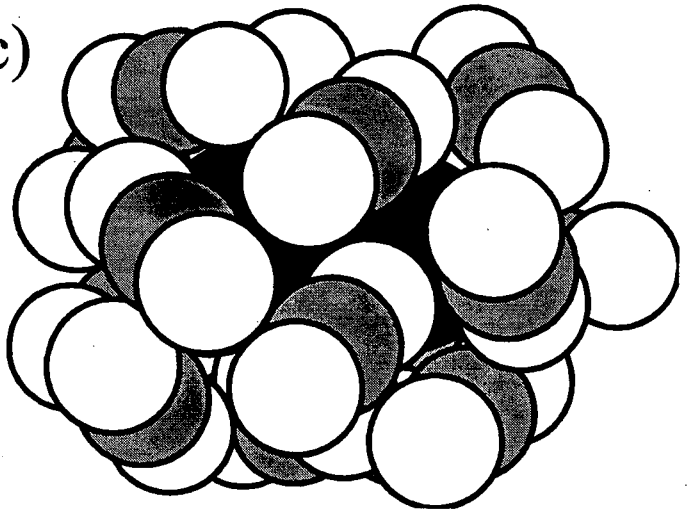
(a)



(b)



(c)



Margulis and Coker[23] also investigated the photodissociation of $I_2^-(CO_2)_8$ and $I_2^-(CO_2)_{16}$ clusters at 720 and 790 nm using a similar nonadiabatic molecular dynamics framework. For the smaller cluster, uncaged fragments are produced in ~500 fs when the electron hops to the more solvated I atom as a result of an electronic transition from the $\tilde{A}'$ state, while caged fragments result when the transition occurs later, allowing solvent rearrangement to trap both I atoms. The larger cluster exhibits complete caging, but at 720 nm, only ~20% recombine within 2 ps, the majority being delayed by 10-25 ps due to trapping in a solvent-separated configuration; at 790 nm, all trajectories are delayed. Vibrational relaxation occurs within ~10 ps after recombination.

The main goal of this study is to follow the evolution of cluster dynamics from the uncaged to fully-caged size regime. Our most important findings include: 1). Short-time (< ~1 ps) changes in the number of solvent molecules around I^-, implying fast solvent rearrangement on the initial $\tilde{A}'$ state, as well as electronic transitions to the $\tilde{X}$ or $\tilde{A}$ states. 2). Time-resolved vibrational relaxation of I_2^- on the $\tilde{X}$ state in $I_2^-(CO_2)_{n\geq 6}$ clusters. 3). Excess numbers of solvent molecules at long time delays (200 ps) around I_2^- $\tilde{X}$ as compared to photofragmentation measurements, implying incomplete solvent evaporation despite rapid relaxation of I_2^-. 4). The presence of a long-lived (>200 ps) solvent-separated I_2^- product in clusters of $I_2^-(CO_2)_{n\geq 9}$, presumably trapped on the $\tilde{A}$ state.

2. Experimental

The apparatus differs slightly from the one described in the previous article,[31] so it is summarized briefly here. $I_2^-(CO_2)_n$ clusters are produced by passing a mixture of 3% CO_2 in Ar at 20 psig over solid I_2 and expanding into vacuum through a pulsed

piezoelectric valve, where the gas is intercepted by a 1.5 keV electron gun. Cluster anions are pulse-extracted into a Wiley-McLaren[35] time-of-flight mass spectrometer where they travel to the laser interaction region. Femtosecond pump pump (780 nm, 80 fs, 150 μJ) and probe (260 nm, 100 fs, 20 μJ) pulses intersect the anions at the focus of a magnetic bottle electron spectrometer,[36] and photoelectrons produced are energy-analyzed by time-of-flight. To increase electron resolution, anions were pulse-decelerated[37] just prior to laser interaction; however, this technique resulted in a loss of ~75% of the electron signal, and so was only applied to clusters with relatively high anion flux [$I_2^-(CO_2)_{n \le 12}$]. However, the slower beam velocity of the heavier anions results in an energy resolution which scales as $m^{-1/2}$, where m is the anion mass. Resolution for 1 eV electrons was 40-50 meV with deceleration, and ~100 meV without deceleration. Typical acquisition times were 80-1000 s per time delay. One-photon (probe only) photoelectron spectra of $I_2^-(CO_2)_n$ clusters (n = 1-10, 12, 14, 16) were also collected, in order to determine the energy shift of I_2^- features due to the presence of CO_2 molecules. Acquisition times for these spectra were 1000-6000 s per cluster.

3. Results

One-photon (probe only) photoelectron spectra of $I_2^-(CO_2)_n$ clusters (n = 1-10, 12, 14, 16) are shown in Fig. 3, and the shift in the vertical detachment energy of the $\tilde{X}$ state relative to bare I_2^- is plotted *vs.* n in Fig 4. A monotonic shift toward lower electron kinetic energy is observed with increasing cluster size, ~150 meV per CO_2 molecule for $I_2^-(CO_2)$, and decreasing to ~50 meV/CO_2 for $I_2^-(CO_2)_{n \ge 10}$. A more systematic study of these trends will be presented in a future publication.[38]

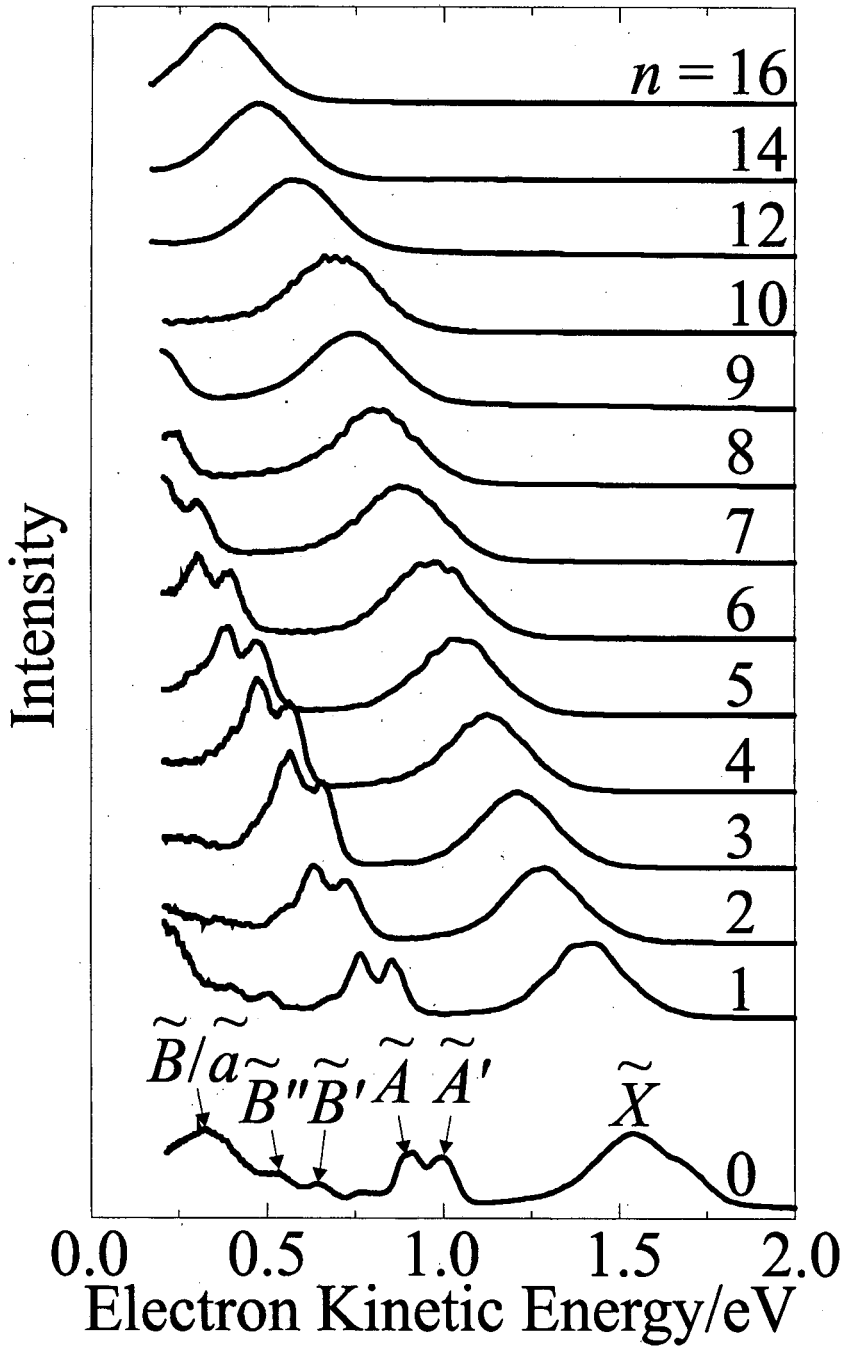


Fig. 3. One-photon photoelectron spectra of $I_2^-(CO_2)_n$ (n = 1-10, 12, 14, 16) clusters. Photon energy = 4.768 eV.

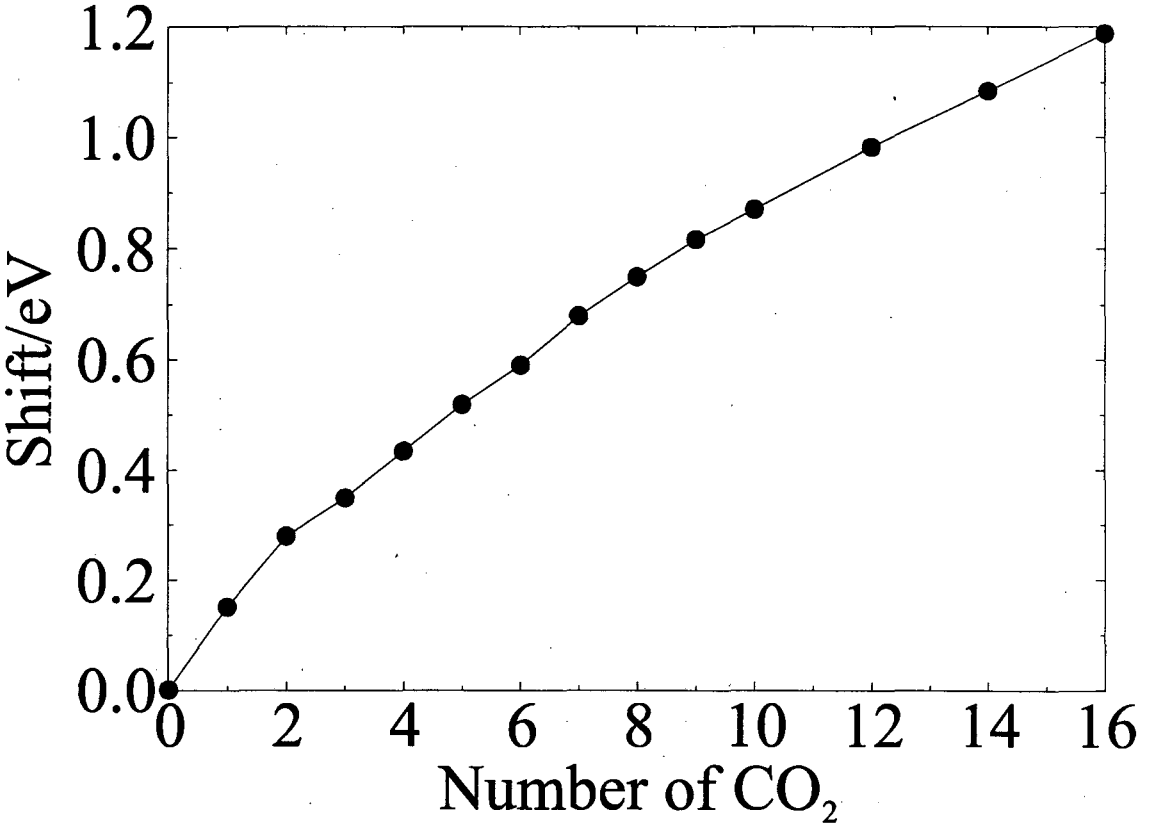


Fig. 4. Solvent shift energy of the I_2^- $\tilde{X}$ state *vs.* n, for $I_2^-(CO_2)_n$ clusters (n = 1-10, 12, 14, 16).

Time-resolved photoelectron spectra for $I_2^-(CO_2)_n$ clusters with n = 4, 6, 9, 12, 14 and 16 are shown in Figs. 5a-c and 6a-c for selected pump-probe delay times. Spectra were taken at many more delays than are shown in the figures; the selected spectra represent the minimum needed to follow the dynamics in these clusters. The spectra consist of peaks of various widths that evolve as a function of time. The maximum time delay recorded for each cluster is 200 ps.

Based on our detailed study of $I_2^-(Ar)_n$ reported in the previous paper,[31] and the mass spectroscopy studies of $I_2^-(CO_2)_n$ photofragmentation by Lineberger and co-workers,[4,5] many of the features in these spectra can be assigned at least on a preliminary basis. This assignment scheme is summarized in Table 1. Feature A, seen at the earliest

times, reflects the transient I_2^- created on the repulsive $\tilde{A}'$ state by the pump pulse; analysis of this feature has been reported in detail elsewhere.[39] Features B, C, and D are relatively sharp peaks that appear by ~1.1 ps. We attribute these to the $I \leftarrow I^-$ transitions, where I^- is solvated by increasing numbers of CO_2 molecules, since for a given cluster they occur at progressively lower electron kinetic energies (eKE). The subscripts 1 and 2, when shown, (n = 4, 6, 9) result from photodetachment to the $^2P_{3/2}$ and $^2P_{1/2}$ states of atomic iodine, respectively. The progression in time and energy for these three features is most apparent for $I_2^-(CO_2)_6$, $I_2^-(CO_2)_9$ and $I_2^-(CO_2)_{12}$, where peak B is most intense at ~200 fs, peak C is most intense at ~400-500 fs, and peak D is most intense by ~800 fs-1.1 ps. A distinct feature C is not evident for the other clusters, but in all cases there is a net shift to lower electron energy from feature B at ~200 fs to feature D at ~800 fs-1.1 ps.

Table 1. Labeling system of features observed in FPES of $I_2^-(Ar)_n$ and $I_2^-(CO_2)_n$ clusters, with corresponding assignments. In cases where two spin-orbit manifolds are visible (peaks A-D), each is labeled with a subscript, e.g. A_1 and A_2, according to decreasing eKE.

Label	Assignment
A	$I_2 \leftarrow I_2^- \ \tilde{A}'$ (transient)
B	$I \leftarrow I^-$ (anomalous state)
C	$I \leftarrow I^-$ (symmetrically solvated)
D	$I \leftarrow I^-$ (dissociated),
	$I \leftarrow I^-$ (solvent-separated I_2^-)
E	$I_2 \ \tilde{X} \leftarrow I_2^- \ \tilde{X}$ $v \approx 0$,
	$I_2 \ \tilde{X} \leftarrow I_2^- \ \tilde{X}$ excited inner turning point (ITP)
F	$I_2 \ \tilde{X} \leftarrow I_2^- \ \tilde{X}$ excited outer turning point (OTP),
	I_2* (higher-lying states) $\leftarrow I_2^- \ \tilde{X}$
G	$I_2 \ \tilde{X} \leftarrow I_2^- \ \tilde{X}$ and $I \leftarrow I^-$ (indistinguishable)

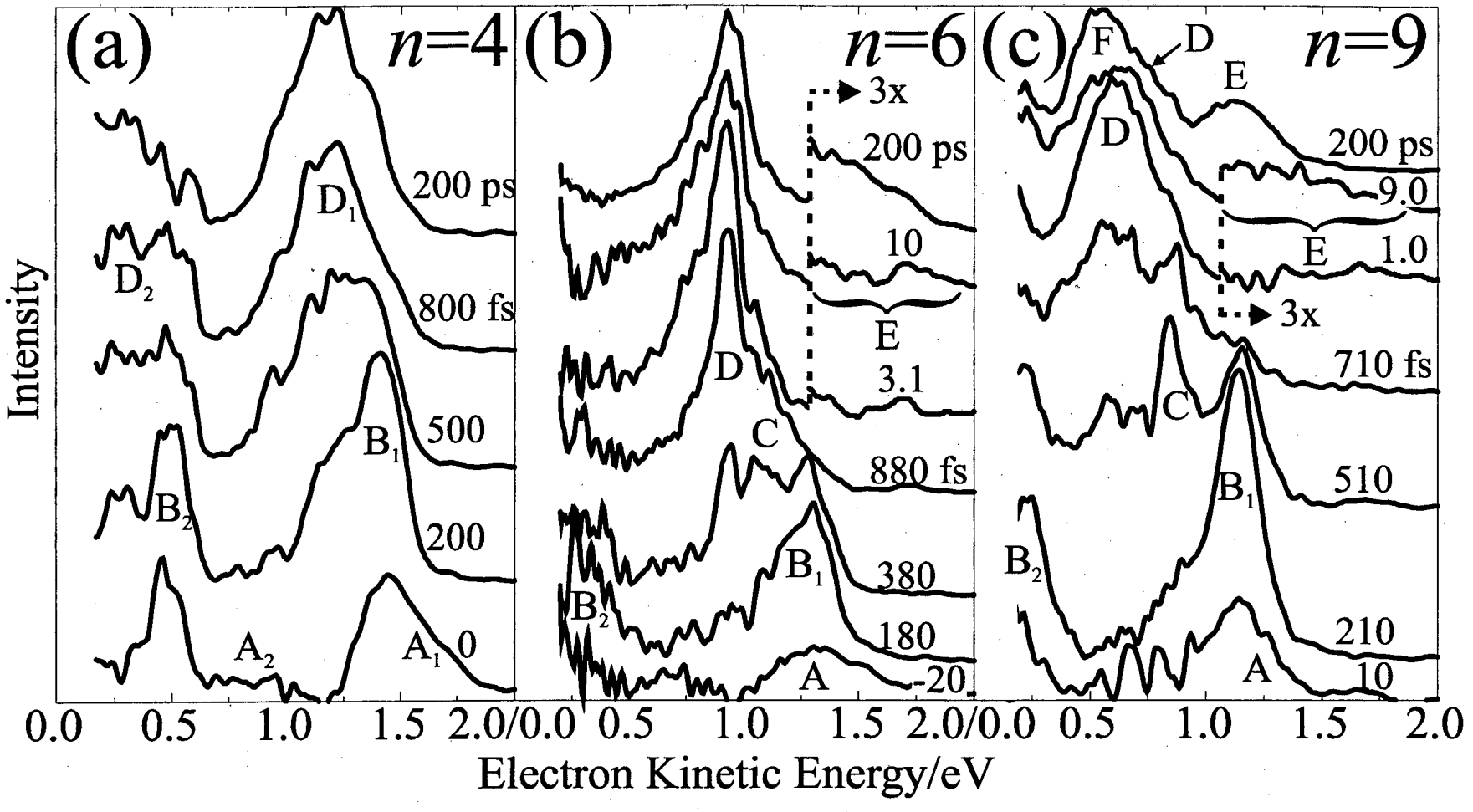
Feature E appears at high electron kinetic energy for all clusters except $I_2^-(CO_2)_4$. The earliest time at which it can be observed drops with increasing cluster size, from ~10

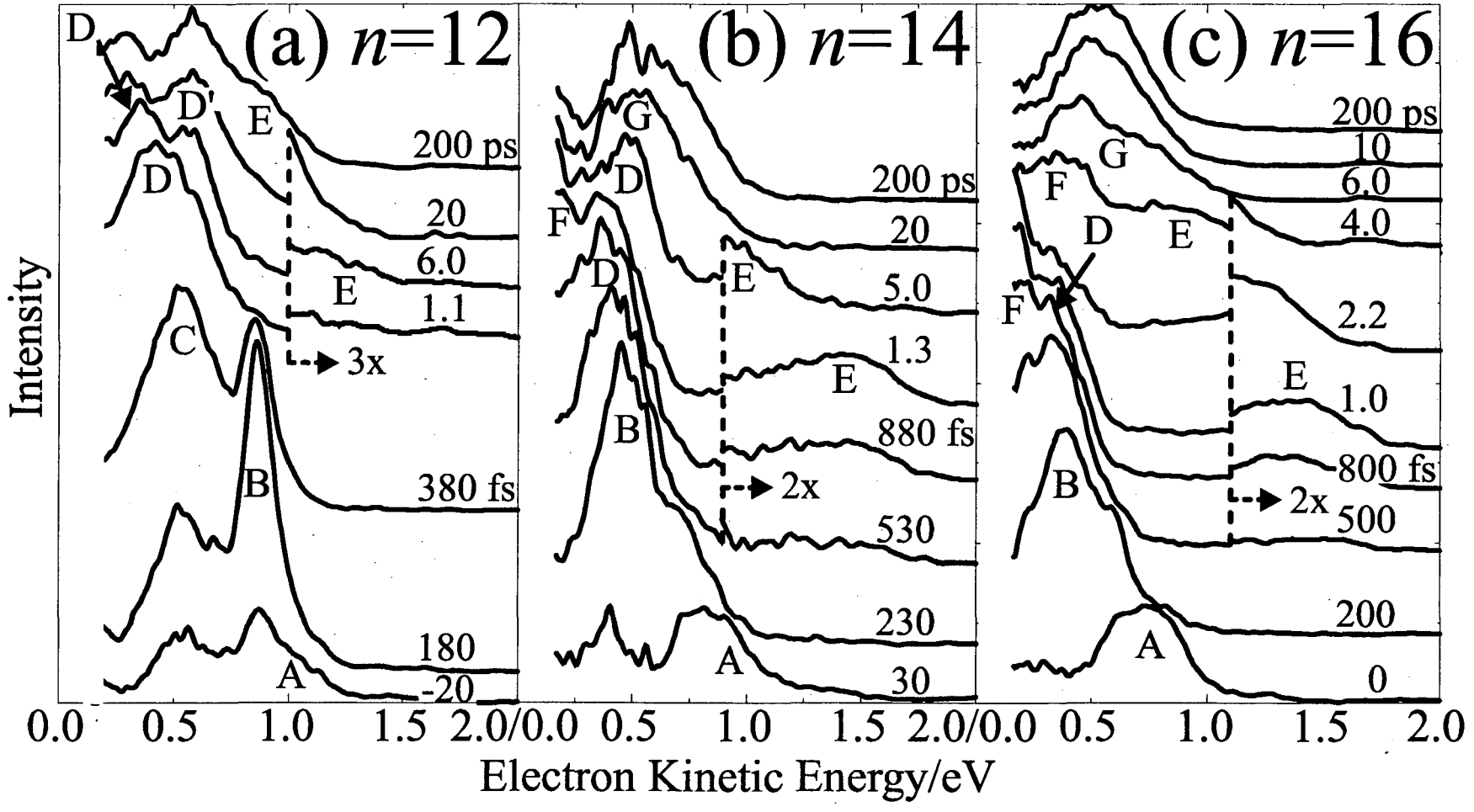
ps for $I_2^-(CO_2)_6$ to ~500 fs for $I_2^-(CO_2)_{14\text{-}16}$. Once this feature appears, it shifts toward lower electron kinetic energy with increasing time, and the degree of shifting is most apparent for the $I_2^-(CO_2)_{14\text{-}16}$ clusters. Based on comparison with our $I_2^-(Ar)_{20}$ results, feature E at early times is attributed to photodetachment from the classical inner turning point region of a vibrationally excited I_2^- wavefunction, resulting from recombination on the $\tilde{X}$ state; the shifting toward lower electron kinetic energy is from subsequent vibrational relaxation of the I_2^-. At time delays > 5 ps for n = 14 and > 4 ps for n = 16, feature E is relabeled feature G to signify that is has merged with lower energy features (see below). In clusters of $n \leq 12$, feature E remains distinguishable from other features in the spectrum at all time delays.

The trends at time delays beyond ~1 ps for n = 9, 12, 14, and 16 clusters at low eKE (< ~1 eV) hint at more complex dynamics that were seen for $I_2^-(Ar)_n$ clusters. The labeling scheme for the spectra in this regime is based on the simulations described in the following sections; it is not obvious by inspection. For $I_2^-(CO_2)_9$, the single feature D at 1.0 ps broadens slightly toward higher eKE, and by 200 ps appears to be a shoulder on the high energy side of a peak labeled F. For $I_2^-(CO_2)_{12}$, feature D splits into two features by 6.0 ps, D (at low eKE) and D′ (at high eKE), with feature E appearing as a shoulder on D′ at 200 ps. In the $I_2^-(CO_2)_{14}$ spectra, feature D begins shifting toward higher eKE after 1.3 ps, and merges with E by 20 ps, at which point the merged feature is labeled G.

Fig. 5. (On next page) FPES of $I_2^-(CO_2)_n$ clusters at selected delay times: (a) $I_2^-(CO_2)_4$, (b) $I_2^-(CO_2)_6$ and (c) $I_2^-(CO_2)_9$. Pump photon energy = 1.589 eV, probe photon energy = 4.768 eV.

Fig. 6. (On page 217) FPES of $I_2^-(CO_2)_n$ clusters at selected delay times: (a) $I_2^-(CO_2)_{12}$, (b) $I_2^-(CO_2)_{14}$ and (c) $I_2^-(CO_2)_{16}$. Pump photon energy = 1.589 eV, probe photon energy = 4.768 eV.


(a)
n=4
(b)
n=6
(c)
n=9
Intensity
Electron Kinetic Energy/eV
0.0
0.5
1.0
1.5
2.0
200 ps
800 fs
500
200
0
10
3.1
880 fs
380
180
-20
9.0
1.0
710 fs
510
210
A
B1
B2
C
D
E
F
D1
D2
A1
A2
3x


(a) n=12
(b) n=14
(c) n=16
Intensity
Electron Kinetic Energy/eV
0.0
0.5
1.0
1.5
2.0
200 ps
20
6.0
1.1
3x
380 fs
180
-20
5.0
1.3
880 fs
2x
530
230
30
10
4.0
2.2
1.0
800 fs
500
200
0
A
B
C
D
D'
E
F
G

A low-energy feature, F, emerges after 880 fs, and drops away again by 20 ps. For $I_2^-(CO_2)_{16}$, D changes from a peak at 500 fs to a shoulder at 1.0 ps on a lower energy feature labeled F. By 4.0 ps, D is gone, leaving E and F, which merge into feature G by 6.0 ps.

The detailed interpretation of this last set of results is possible only by comparison with simulations described below. These will demonstrate that peak F has a similar origin as in the $I_2^-(Ar)_n$ spectra: namely, photodetachment from the classical outer turning point of vibrationally excited I_2^- to the ground state of I_2, and from vibrationally relaxed I_2^- to the low-lying excited states of I_2. The simulations also suggest that feature D in clusters of $I_2^-(CO_2)_{n \geq 9}$ [D′ in $I_2^-(CO_2)_{12}$] is due to formation of a long-lived, solvent-separated state of I_2^- within the cluster.

4. Analysis

Simulations of the photoelectron spectra were performed in a manner similar to that for the $I_2^-(Ar)_n$ clusters,[31] using a combination of previously-measured photoelectron spectra to simulate the I^- features, and calculated photoelectron spectra to model vibrationally excited I_2^- features. Details of the methods used can be found in the previous paper,[31] but a brief overview of considerations unique to the $I_2^-(CO_2)_n$ simulations are discussed below.

The number of solvent molecules around I^-, termed "$\langle n_{I^-} \rangle$," was determined from previously-measured solvent shifts of $I^-(CO_2)_n$ clusters.[40-42] The shifts are much larger per CO_2 molecule than per Ar atom, ~170 meV for $I^-(CO_2)$, decreasing to ~40 meV for $I^-(CO_2)_{n \geq 10}$. For analysis of short-time (< ~1 ps) spectra, the eKE of features B, C and D have been converted into $\langle n_{I^-} \rangle$ (using linear interpolation to calculate a fractional value when the eKE lay between measured shifts) and are presented in Table 2. Included in this

table are estimates of $<n_I^->$ for the anomalous and normal charge-switching states, based on calculated structures,[18,22,34] and assuming no loss of CO_2 molecules. An estimate for a symmetrically-solvated structure, which would occur at later time delays on the $\tilde{A}'$ state, is also shown, obtained by averaging the anomalous and normal values. In clusters of $I_2^-(CO_2)_{n \geq 9}$, there is evidence that a solvent-separated I_2^- structure ("SS I_2^-") is at least partially responsible for feature D [D′ in the case of $I_2^-(CO_2)_{12}$]. This structure appears spectroscopically as solvated I^-, because the I^- is surrounded by CO_2 molecules, with the I atom located outside the solvent shell. This interpretation is explored in detail in the Discussion.

Table 2. Calculated number of solvent molecules $<n_I^->$ for features B, C and D in $I_2^-(CO_2)_n$ clusters from FPES. Estimated $<n_I^->$ for anomalous ($\tilde{A}'$) and normal ($\tilde{X}$, $\tilde{A}$) charge-switching states, and $<n_I^->$ for a symmetric solvent configuration on the $\tilde{A}'$ state, derived from model structures.[18]

Parent	FPES			Model estimates		
Cluster	B	C	D	Anomalous	Symmetric	Normal
4	1.4	-	2.5	3.0	3.5	4.0
6	2.1	3.4	4.4	3.0	4.5	6.0
9	3.0	5.0	6.7	3.0	6.0	9.0
12	4.8	7.2	8.0	5.0	7.0	9.0
14	7.8	-	8.2	9.0	9.5	10.0
16	8.4	-	8.9	10.0	10.0	10.0

For simulating FPES spectra, actual $I^-(CO_2)_n$ photoelectron spectra were used,[41] rather than the shifted I^- spectra used for $I^-(Ar)_n$, because the $I^-(CO_2)_n$ spectra contain prominent progressions arising from CO_2 vibrational modes. Normalization of the spectra was achieved by scaling the integral of the $I\ ^2P_{3/2} \leftarrow I^-\ ^1S_0$ transition to that of the bare I^- spectrum [2.0, relative to 1.0 for the $I_2\ \tilde{X} \leftarrow I_2^-\ \tilde{X}$ (v = 0) transition].[31]

Solvated I_2^- $\tilde{X}$ was modeled using an empirically determined distribution of vibrational levels, designated by an average level "<v>," and average number of solvent molecules "$\langle n_X \rangle$." The solvent shift energies from the one-photon photoelectron spectra of $I_2^-(CO_2)_n$ clusters measured in this work (Fig. 4) were employed to shift the simulated spectra. The method of generating spectra, and the relative intensities of transitions, were identical to those used for $I_2^-(Ar)_n$ clusters, including the energy-dependent scaling function $f(E)$ for the $\tilde{X} \leftarrow \tilde{X}$ transition.[31]

The most important task in fitting the solvated I_2^- $\tilde{X}$ state features is determining $\langle n_X \rangle$ and <v>. At long time delays, when <v> is fairly small (<~5), simultaneous determination of $\langle n_X \rangle$ and <v> from the spectrum is possible by simulating feature E (or G). In this situation, the feature arises from a compact I_2 $\tilde{X} \leftarrow I_2^-$ $\tilde{X}$ transition whose shape depends sensitively on <v>, while the overall eKE of the transition is governed by $\langle n_X \rangle$. At larger <v>, the $\tilde{X} \leftarrow \tilde{X}$ transition splits into two distinct energy regions arising from different segments of the I_2^- vibrational wavefunction. Near the classical inner turning point (ITP) of the I_2^- potential, the transition exhibits a broad range of eKE's, and the maximum eKE increases rapidly with <v>. Near the classical outer turning point (OTP), the transition is much narrower and intense, occurring at lower eKE, and its eKE is relatively insensitive to <v> over a wide range (~10-30), giving an indication of $\langle n_X \rangle$. The $\tilde{X} \leftarrow \tilde{X}$ ITP transition, lying at the highest eKE in the spectrum, corresponds to feature E at short time delays; however, as its eKE is dependent on $\langle n_X \rangle$ and <v>, both quantities cannot be simultaneously determined from this feature. In the $I_2^-(Ar)_n$ clusters, the $\tilde{X} \leftarrow \tilde{X}$ OTP transition, as well as transitions from I_2^- $\tilde{X}$ to

higher-lying I_2 states ($\tilde{A}'$, $\tilde{A}$, $\tilde{B}'$, $\tilde{B}''$, $\tilde{a}$, $\tilde{a}'$ and $\tilde{B}$), collectively referred to as the I_2^* $\leftarrow \tilde{X}$ transitions, occur at lower eKE where they are denoted by feature F. This feature was used to determine $\langle n_X \rangle$, as the eKE's of the transitions are all relatively insensitive to $\langle v \rangle$.[31] In $I_2^-(CO_2)_n$ clusters, however, the transitions either are not accessible at the probe wavelength due to large solvent shifts, or they are masked by overlapping I^- features, making accurate determination of $\langle n_X \rangle$ difficult.

In these cases [primarily at short time delays in $I_2^-(CO_2)_{14}$ and $I_2^-(CO_2)_{16}$ clusters], simulations using several values of $\langle n_X \rangle$ were performed. Several criteria were then considered in determining a "best" $\langle n_X \rangle$ time progression for the cluster. The value of $\langle n_X \rangle$ at long time delays, where it could be determined with more confidence, served as a primary constraint. It was assumed that $\langle n_X \rangle$ decreased monotonically with time delay, and that changes from one spectrum to another were not abrupt ($\leq$ 1 CO_2 molecule). The correspondence between the experimental spectra and simulations using different values of $\langle n_X \rangle$ was not equally good, further limiting the choices of $\langle n_X \rangle$. A final decision rested on careful comparison of $\langle n_X \rangle$ in neighboring clusters, and the photofragmentation averages. These issues are covered in detail in the Discussion.

Populations of the I^- and I_2^- $\tilde{X}$ state contributions, indicated by "P_{I^-}" and "P_X," respectively, were determined from the intensities of simulated spectral features, weighted by their relative cross-sections. P_{I^-} and P_X sum to unity for all spectra.

Simulation parameters are summarized in a series of graphs in Figs. 7a-b (P_X and $\langle v \rangle$ *vs.* time), and 8a-b ($\langle n_X \rangle$ and $\langle n_{I^-} \rangle$ *vs.* time). Included in these figures are data from $I_2^-(Ar)_{20}$, and photofragmentation values indicated as detached points in the P_X, $\langle n_X \rangle$ and $\langle n_{I^-} \rangle$ *vs.* time graphs. Although several time delays have been simulated to follow the

dynamics in each cluster, only a selected number are shown to illustrate the changes taking place. These are presented in Figs. 9 (n = 4), 10a-f (n = 6, 9 and 12), 12a-c (n = 14) and 13a-d (n = 16), and each graph includes curves representing the solvated I^- and I_2^- $\tilde{X}$ contributions, the total simulated spectrum, and the experimental spectrum.

5. Discussion

Cluster spectra are treated in three sections, grouped by similar dynamics: $I_2^-(CO_2)_4$, which shows relatively simple solvated I^- formation, and no I_2^-; $I_2^-(CO_2)_6$, $I_2^-(CO_2)_9$ and $I_2^-(CO_2)_{12}$, which exhibit more complex I^- dynamics, and increasing amounts of I_2^- recombination and vibrational relaxation; and $I_2^-(CO_2)_{14}$ and $I_2^-(CO_2)_{16}$, where I^- dynamics are rapidly obscured by I_2^- features, which exhibit extensive vibrational relaxation. For reference in the following discussions, the fraction of I_2^- products measured in the photofragmentation experiments[4] are: 0.03 in $I_2^-(CO_2)_4$, 0.29 in $I_2^-(CO_2)_6$, 0.70 in $I_2^-(CO_2)_9$, 0.85 in $I_2^-(CO_2)_{12}$, 0.95 in $I_2^-(CO_2)_{14}$, and 1.00 in $I_2^-(CO_2)_{16}$. Thus, there are essentially no I_2^- products in $I_2^-(CO_2)_4$, and no I^- products in $I_2^-(CO_2)_{14\text{-}16}$. After discussion of each group of clusters, trends across cluster size are summarized, and comparisons are made to previous experimental and theoretical work. Parallels with $I_2^-(Ar)_n$ clusters are made in the final section.

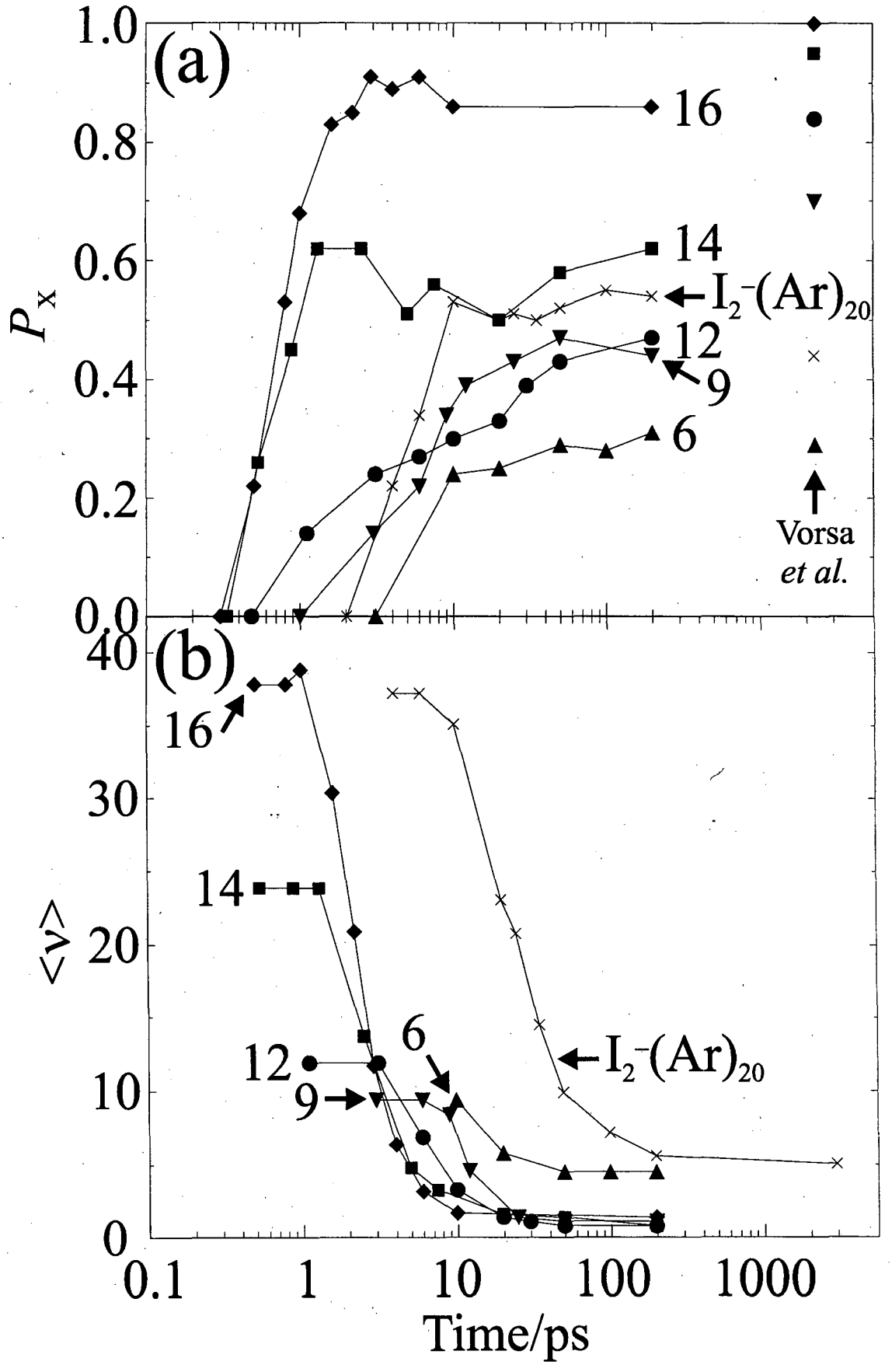
5.1. $I_2^-(CO_2)_4$

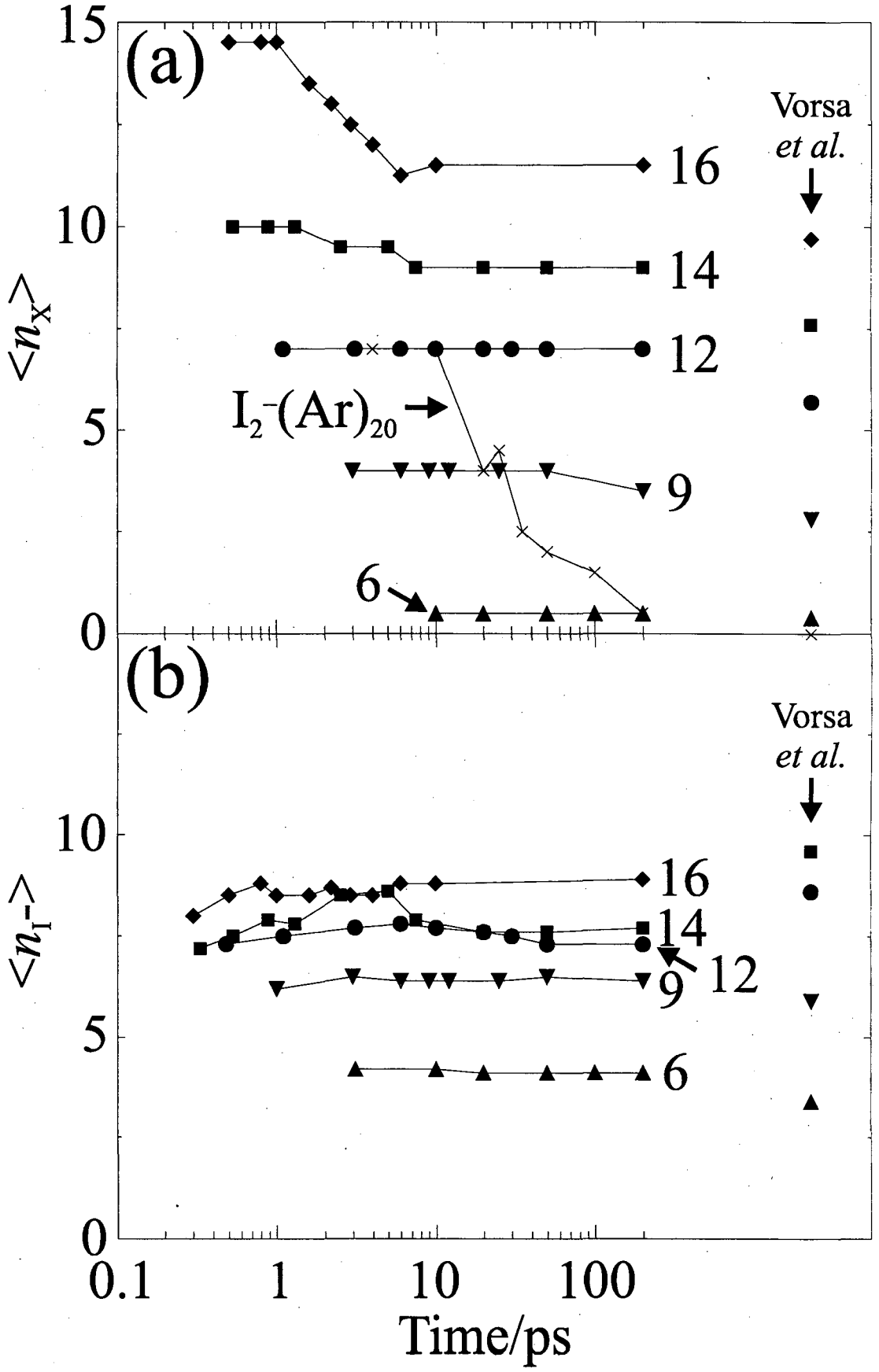
In $I_2^-(CO_2)_4$, I^--based products dominate in the photofragmentation experiments, and no I_2^- features are observed in the FPES spectra. This cluster is also predicted to possess an initial solvent configuration which is fairly symmetric, with three of the four solvent molecules around the I_2^- waist.

We begin by focusing on the short-time dynamics. Feature B, peaking at ~200 fs, represents the earliest observable I^- signal, when the I^- and I are not very far apart and the system is presumably still on the $\tilde{A}'$ state. For this feature, $\langle n_{I^-} \rangle = 1.4$, significantly smaller than the total number in the cluster, which indicates substantial movement of the I^- away from the cluster by this time delay. By the time feature D fully forms at 800 fs, $\langle n_{I^-} \rangle$ is larger (2.5), suggesting a more heavily solvated environment around I^-. The increase could be due to either rearrangement on the $\tilde{A}'$ state into a symmetric solvent configuration, or a transition to the normal charge-switching $\tilde{X}$ or $\tilde{A}$ state which returns the electron to the original, more solvated I atom. These alternatives are represented in Table 2 as the "symmetric" and "normal" estimates, respectively. However, because the $\langle n_{I^-} \rangle$ of D is smaller than either of these estimates, it is not possible to distinguish among these possibilities from the spectra.

Fig. 7. (On next page) Simulation parameters for $I_2^-(CO_2)_n$ clusters: (a) Population of the I_2^- $\tilde{X}$ state (P_X) *vs.* time, (b) Average vibrational level of the I_2^- $\tilde{X}$ state ($\langle v \rangle$) *vs.* time. Legend: upward-pointing triangles = $I_2^-(CO_2)_6$, downward-pointing triangles = $I_2^-(CO_2)_9$, circles = $I_2^-(CO_2)_{12}$, squares = $I_2^-(CO_2)_{14}$, diamonds = $I_2^-(CO_2)_{16}$, crosses = $I_2^-(Ar)_{20}$. Detached points indicate values from Vorsa *et al.*[10]

Fig. 8. (On page 224) Simulation parameters for $I_2^-(CO_2)_n$ clusters: (a) Average number of CO_2 molecules in the I_2^- $\tilde{X}$ state ($\langle n_X \rangle$) *vs.* time, (b) Average number of CO_2 molecules in the I^- fragment ($\langle n_{I^-} \rangle$) *vs.* time. Legend is identical to Fig. 7. Detached points indicate values from Vorsa *et al.*[10]


(a)
P_x
1.0
0.8
0.6
0.4
0.2
0.0
16
14
$I_2^-(Ar)_{20}$
12
9
6
Vorsa
et al.
(b)
$\langle v \rangle$
40
30
20
10
0
16
14
12
9
6
$I_2^-(Ar)_{20}$
0.1
1
10
100
1000
Time/ps


(a)
(b)
Vorsa
et al.
16
14
12
9
6
I2-(Ar)20
<nx>
<nI->
15
10
5
0
0.1
1
10
100
Time/ps

The FPES of $I_2^-(CO_2)_4$ undergo little change after 800 fs, demonstrating the extremely fast nature of the reaction: essentially the unencumbered photodissociation of I_2^-. To illustrate the composition of the products, the 200 ps spectrum was simulated using a distribution of $I^-(CO_2)_n$ spectra, shown in Fig. 9. This distribution does not match the photofragmentation data exactly, with $<n_{I^-}>$ (2.6) somewhat higher than observed in these experiments ($<n_{I^-}> = 1.9$). This discrepancy points to possibly vibrationally excited product which has not dissipated its excess energy through solvent evaporation by 200 ps.

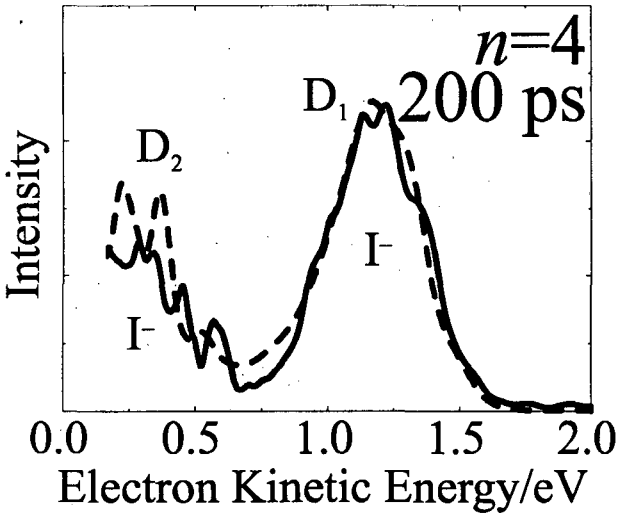


Fig. 9. Simulated spectra of $I_2^-(CO_2)_4$ at 200 ps. Experimental (thick solid line) and simulated (thick dotted line) data are shown. Only I^- contributions were required to reproduce the experimental spectrum.

5.2. $I_2^-(CO_2)_6$, $I_2^-(CO_2)_9$ and $I_2^-(CO_2)_{12}$

These intermediate-sized clusters exhibit both solvated I^- and I_2^- features in their spectra at long time delays, consistent with the photofragmentation experiments. They are also predicted[18,22,34] to have highly asymmetric initial solvent configurations, that is, with the more negative I atom strongly solvated, and other I atom significantly less solvated. This asymmetry is exhibited in the short-time dynamics as three distinct solvated I^- features (at different eKE's) in the first ~1.0 ps after photodissociation. These short-time

dynamics will be discussed first, and then simulations of the I_2^- dynamics will be covered for each cluster individually.

As shown in Table 2, the difference in $<n_I^->$ between the anomalous and normal estimates is fairly substantial for these clusters, allowing the charge-switching state responsible for the $<n_I^->$ of feature B to be determined unambiguously. In all cases, $<n_I^->$ is closest to the anomalous estimate, meaning that the negative charge is initially on the less-solvated I atom. The increases in $<n_I^->$ for features C and D indicate more heavily solvated environments around I^-, but the presence of two distinct features, one of which is short-lived (C), and the other of which persists for several ps (D), suggests the existence of two, possibly coupled, mechanisms. As for $I_2^-(CO_2)_4$ above, both features could be due to either rearrangement on the $\tilde{A}'$ state into a symmetric solvent configuration, or a transition to the normal charge-switching $\tilde{X}$ or $\tilde{A}$ state which returns the electron to the original, more solvated I atom. Some CO_2 molecules probably escape from the cluster during the first ~1 ps, making the estimates for $<n_I^->$ upper limits on the observed value of $<n_I^->$ corresponding to each structure (symmetric or normal). Bearing this in mind, in each cluster there is a reasonable correspondence between the $<n_I^->$ of feature C and the symmetric estimate, and between the $<n_I^->$ of feature D and the normal estimate. This implies a mechanism where the solvent molecules first rearrange into an energetically favorable, symmetric configuration on the $\tilde{A}'$ state (feature C), which maximizes the coupling with the $\tilde{X}$ and $\tilde{A}$ states. After a transition to either of these normal charge-switching states, the solvent rapidly rearranges again into a configuration where I^- is heavily solvated (feature D). Theoretical models must be examined to confirm these assignments; the work of Faeder *et al.*[22] and Margulis *et al.*[23] will be considered.

The Faeder *et al.* model predicts for $I_2^-(CO_2)_9$ and $I_2^-(CO_2)_{12}$ [$I_2^-(CO_2)_6$ was not considered] that the solvent rearranges to a symmetric configuration within 500 fs, followed by transitions to the $\tilde{A}$ or $\tilde{X}$ states at later time delays. The electronic transitions are much slower in $I_2^-(CO_2)_9$ (~2 ps) than $I_2^-(CO_2)_{12}$ (~700 fs), and is dominated by the $\tilde{A} \leftarrow \tilde{A}'$ transition in this time range. The solvent rearrangement step is consistent with the assignment of feature C, which appears with maximum intensity at ~500 fs in both clusters. The $\tilde{A} \leftarrow \tilde{A}'$ transition corresponds to feature D in $I_2^-(CO_2)_{12}$, but in $I_2^-(CO_2)_9$ the transition occurs considerably more slowly in the model than observed experimentally. The discrepancy may by symptomatic of the predicted slow transitions to the $\tilde{X}$ state from both the $\tilde{A}'$ and $\tilde{A}$ states, which are discussed below in comparing the appearance of the $\tilde{X}$ state in this model with experiment.

The model of Margulis *et al.* investigated $I_2^-(CO_2)_8$ photodissociation at 720 nm, which, while not identical to any of the clusters studied by FPES, is worth examining. For ~50% of the trajectories, the electron returns to the original I atom (i.e., normal charge-switching state) in ~500 fs, leading to rapid dissociation. The remaining trajectories undergo solvent rearrangement on the $\tilde{A}'$ state, with a transition to the $\tilde{X}$ or $\tilde{A}$ state occurring within 30 ps, resulting in I_2^- recombination. This description presents a significantly different picture of the dynamics than the Faeder *et al.* study. The ~500 fs transition to the $\tilde{A}$ or $\tilde{X}$ state would correspond to feature C, which appears at the same time, while the slower rearrangement of solvent molecules on the $\tilde{A}'$ state would be indicated by feature D. However, the model predicts that the $\tilde{X}$ / $\tilde{A}$ state trajectories lead to rapid dissociation, implying that C remains in the spectrum as an $I^-(CO_2)_n$ product, while in fact it vanishes by 1.0 ps. A possible explanation for this disappearance might be

a subsequent solvent rearrangement on the $\tilde{X}$ / $\tilde{A}$ state, increasing $\langle n_{I^-} \rangle$ so that the dissociating I^- signal overlaps with feature D, the trapped I_2^- $\tilde{A}'$ signal, at longer time delays.

We are inclined to favor the Faeder *et al.* model since it agrees with the arguments made at the beginning of this section, and, despite its shortcomings, offers a slightly simpler explanation than the Margulis *et al.* model.

Simulations for individual clusters will now be discussed.

The FPES of $I_2^-(CO_2)_6$ were simulated between 10 ps (the earliest time delay at which an indisputable E feature was identified) and 200 ps. Over this time period, P_X grows slowly from 0.24 at 10 ps to 0.31 at 200 ps (Fig. 7a), while <v> decreases in step with the rise of P_X (Fig. 7b) from 9.4 to 4.5; $\langle n_X \rangle$ and $\langle n_{I^-} \rangle$ are virtually unchanged after 10 ps (Figs. 8a-b). $I_2^-(CO_2)_6$ displays the slowest appearance of the $\tilde{X}$ state in all the clusters studied here, and the initial value of <v> is also smaller than that seen in any other cluster. Since I_2^- must enter the $\tilde{X}$ state from either the $\tilde{A}$ or $\tilde{A}'$ state with ~1 eV of internal energy (the difference in bond energies), the low initial value of <v>, corresponding to only 130 meV of internal energy, must indicate an average of clusters over a wide range of vibrational levels. This idea is reflected in the large width of the distribution (standard deviation $\sigma_v \approx$ <v>), which narrows in the larger clusters ($n \geq 14$) at early time delays. The growth of P_X is also correlated with the decrease in <v>, as shown by comparing Figs. 7a and 7b. Thus, the rate-limiting step in vibrational relaxation is the influx of population to $\tilde{X}$ state.

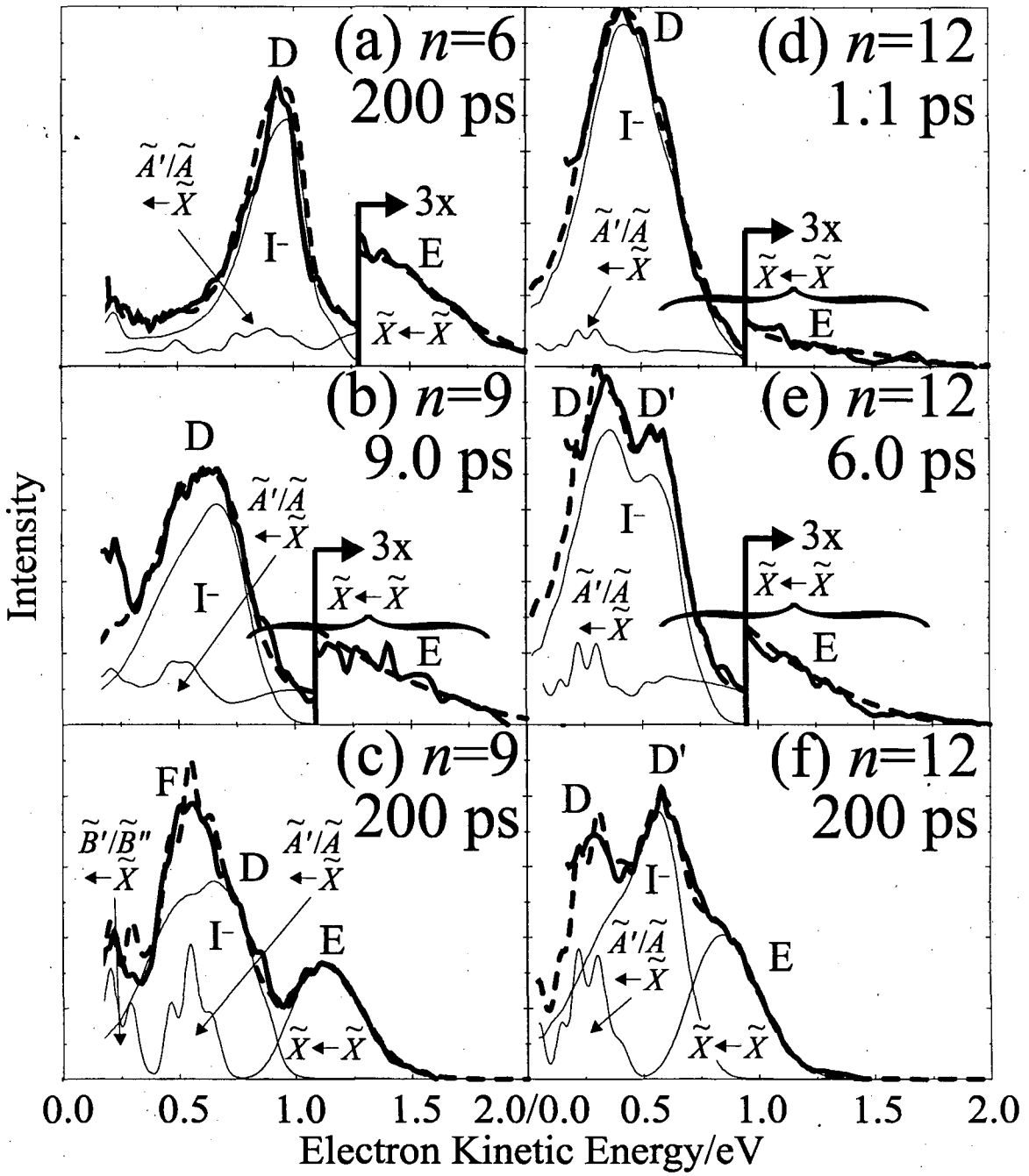


Fig. 10. Simulated spectra of (a) $I_2^-(CO_2)_6$ at 200 ps, (b) $I_2^-(CO_2)_9$ at 9.0 ps, (c) $I_2^-(CO_2)_9$ at 200 ps, (d) $I_2^-(CO_2)_{12}$ at 1.1 ps, (e) $I_2^-(CO_2)_{12}$ at 6.0 ps, and (f) $I_2^-(CO_2)_{12}$ at 200 ps. Experimental (thick solid line) and simulated (thick dotted line) data are shown. Thin lines indicate simulated I^- and I_2^- $\tilde{X}$ -based CO_2 fragment contributions.

As the changes in the spectra after 10 ps were not dramatic, only the 200 ps simulation is shown (Fig. 10a), which illustrates the features encountered in all the

spectra. The simulation parameters (P_X = 0.31, <v> = 4.5, <n_X> = 0.5, <n_{I^-}> = 4.1) indicate a spectrum dominated by solvated I^-, which almost entirely accounts for feature D. Solvated I_2^- $\tilde{X}$ generates a fairly uniform, low-intensity signal over the entire range of observed electron kinetic energies, 0-2 eV, with a distinctive, sloping shape in the region of the E feature, arising from the $\tilde{X} \leftarrow \tilde{X}$ transition, and a slight increase at 750 meV due to the $\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$ transitions. Because of the broad appearance of E, there was some latitude in the choice of <n_X> (0.5-1.5), which influences the values of the other parameters, but the changes are not substantial. The photofragmentation average of 0.4 is comparable to the low end of this range. As for $I_2^-(CO_2)_4$, <n_{I^-}> is larger than the photofragmentation average (3.4), which points to vibrational excitation and incomplete evaporation in the solvated I^- fragment at 200 ps. <v> is also the largest of any cluster at 200 ps, suggesting inefficient energy transfer to the small number of CO_2 molecules, i.e., 0 or 1, present in the cluster. In larger clusters, with a larger number of CO_2 molecules remaining, <v> attains values as low as 0.8.

In $I_2^-(CO_2)_9$, the spectra were simulated from 1.0 to 200 ps. More significant changes in the shape of feature E are observed than in $I_2^-(CO_2)_6$, so two time delays (9.0 and 200 ps) are shown in Figs. 10b-c to illustrate the changes. The 9.0 ps spectrum (P_X = 0.34, <v> = 8.4, <n_X> = 4.0, <n_{I^-}> = 6.4) contains an $\tilde{X}$ state signal which appears very similar to that seen in the $I_2^-(CO_2)_6$ simulation: intensity fairly uniformly distributed over 0-2 eV, with a sloped structure in the vicinity of feature E ($\tilde{X} \leftarrow \tilde{X}$), and a peak at 500 meV ($\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$). Feature D is almost entirely accounted for by solvated I^-. As with $I_2^-(CO_2)_6$, there is some uncertainty in <n_X> (±1.0) which alters the other parameters slightly, but the simulation is essentially unchanged. By contrast, the 200 ps spectrum (P_X

= 0.44, <v> = 1.1, $\langle n_X \rangle$ = 3.5, $\langle n_{I^-} \rangle$ = 6.4) exhibits a much more structured $\tilde{X}$ state signal, which is peaked at 1.15 eV ($\tilde{X} \leftarrow \tilde{X}$, corresponding to E), ~550 meV ($\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$, feature F) and ~250 meV ($\tilde{B}'/\tilde{B}'' \leftarrow \tilde{X}$). Solvated I^- still accounts for the majority of the intensity of D; it is broadened and diminished relative to 9.0 ps, but otherwise unchanged. $\langle n_X \rangle$ is well-defined at this time delay (< ±0.5), due to a compact and easily distinguished E feature.

P_X (Fig. 7a) increases more rapidly than in $I_2^-(CO_2)_6$, essentially reaching its final value of 0.44 by 25 ps, while <v> (Fig. 7b) tracks the rise in P_X, falling from 9.4 to 1.4 by this time delay. There is little change to either $\langle n_X \rangle$ or $\langle n_{I^-} \rangle$ after 1.0 ps (Figs. 8a-b). As for $I_2^-(CO_2)_6$, the earliest value of <v> is fairly small with a wide distribution of levels ($\sigma_v \approx$ <v>), and the correlated changes between P_X and <v> probably indicate that the fraction of highly vibrationally excited molecules is small at any given time, with relaxation limited by the initial transition into the $\tilde{X}$ state.

Both $\langle n_X \rangle$ and $\langle n_{I^-} \rangle$ are ~0.5 CO_2 molecules larger than the photofragmentation averages of 2.8 and 5.9, respectively, which indicate that some energy still resides in CO_2 vibrational modes. This is despite the fact that the 200 ps value of <v> is much lower than in $I_2^-(CO_2)_6$, indicating more complete vibrational relaxation. Thus, energy is efficiently transferred from I_2^- to CO_2 modes, but much less efficiently to CO_2 evaporation. This effect will become even more pronounced in larger clusters, as the number of CO_2 modes increases.

At 200 ps, the spectrum is dominated by solvated I^-, with P_X = 0.44 far smaller than the photofragmentation value of 0.70. Together with stronger spectroscopic evidence to be described in $I_2^-(CO_2)_{12}$, this discrepancy is explained by the trapping of part of the

I_2^- population into an SS I_2^- structure, which appears as solvated I^- spectroscopically. Approximately half the I^- signal must be due to SS I_2^- in order to account for the small value of P_X as compared to the photofragmentation experiment. SS I_2^- is predicted by Faeder *et al.*[22] to contain more CO_2 molecules than would be indicated by the I^- solvent shift, since some CO_2's surround the neutral I. However, in the case of $I_2^-(CO_2)_9$, it appears that there is little difference between the I^- and SS I_2^- spectral signals, other than a ~50% increase in the width of the distribution between 3.0 and 25 ps, indicating possible differences in $\langle n_I^- \rangle$ which are not discernible within the resolution of the spectra.

$I_2^-(CO_2)_{12}$ clusters were simulated from 1.1 to 200 ps. Like $I_2^-(CO_2)_9$, the spectra exhibit significant changes during this time interval; three time delays (1.1, 6.0 and 200 ps) are shown to illustrate them in Figs. 10d-f. At 1.1 ps, the spectrum (P_X = 0.14, $\langle \nu \rangle$ = 11.9, $\langle n_X \rangle$ = 7.0, $\langle n_I^- \rangle$ = 7.5) is almost entirely due to solvated I^-; the small amount of solvated I_2^- $\tilde{X}$ accounts for the small tail comprising E, with a negligible amount of intensity present in the vicinity of D. The distribution of $I^-(CO_2)_n$ contributions is strongly peaked near the average value. The spectrum at 6.0 ps (P_X = 0.27, $\langle \nu \rangle$ = 6.9, $\langle n_X \rangle$ = 7.0, $\langle n_I^- \rangle$ = 7.8) consists of a more substantial I_2^- $\tilde{X}$ state, resembling the one simulated for $I_2^-(CO_2)_9$ at 9.0 ps, with a sloped structure accounting for E ($\tilde{X} \leftarrow \tilde{X}$), and increased intensity at ~200 meV ($\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$). The I^- distribution is double-peaked at n = 7 and n = 9, though with an average number which is roughly the same as in the 1.1 ps spectrum. These two peaks largely account for features D and D′; the $\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$ transition makes a minor contribution to D. The 200 ps spectrum (P_X = 0.47, $\langle \nu \rangle$ = 0.8, $\langle n_X \rangle$ = 7.0, $\langle n_I^- \rangle$ = 7.3) exhibits strongly peaked $\tilde{X}$ state signals at 850 meV ($\tilde{X} \leftarrow \tilde{X}$, feature

E) and ~200 meV ($\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$, accounting for half of feature D's intensity), and an I^- signal which is much less double-peaked, the majority of the intensity occurring at $n = 7$. The I^- contribution accounts for the entirety of D´, and half the intensity of D.

The double-peaked I^- distribution is a fairly direct indication of the presence of the SS I_2^- structure, for the following reasons. While the Vorsa *et al.* study[4,10] observed no $I^-(CO_2)_n$ photofragments with $n < 8$, feature D´ is accounted for almost entirely by $I^-(CO_2)_{n \leq 7}$. Feature D, reproduced mainly by n = 8-10 (as well as the $\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$ transitions at later time delays), represents an I^- distribution much closer to the photofragmentation average (8.6). Even the overall $\langle n_{I^-} \rangle$ at all time delays (7.3-7.8) is smaller than the photofragmentation average, while in smaller clusters, it was always observed to be larger. An excess of CO_2 molecules over the photofragmentation average is easy to account for in terms of incomplete solvent evaporation, but the only realistic way to account for too few CO_2 molecules is the SS I_2^- structure, which is predicted to contain more CO_2 molecules than are apparent from the FPES signal.[22] In addition, the 200 ps spectrum is dominated by the solvated I^- features, with P_X (0.47) far short of the photofragmentation result (0.84), as in $I_2^-(CO_2)_9$, indicating a major additional I^--like component.

Therefore, we propose that feature D´, simulated principally by $I^-(CO_2)_{n \leq 7}$, indicates the SS I_2^- structure, while D, represented mainly by $I^-(CO_2)_{n \geq 8}$, indicates "normal" dissociated I^-. To illustrate the time-resolved changes in the I^- distribution more clearly, the simulated populations of individual $I^-(CO_2)_n$ contributions are plotted in Fig. 11 as a series of histograms. At 1.1 ps, a Gaussian-like distribution is observed from n = 5 to 9, with $\langle n_{I^-} \rangle$ = 7.5. By 6.0 ps, however, the double-peaked structure with major

components $n = 7$ and 9 is clearly apparent. Through 200 ps, the $n \leq 7$ contribution remains essentially constant near 0.41, while the $n \geq 8$ component diminishes to only 0.12. Assuming that all of the SS I_2^- eventually recombines on the $\tilde{X}$ state, P_X may be added to the SS I_2^- population to obtain a predicted total of 0.88, very close to the photofragmentation value (0.84).

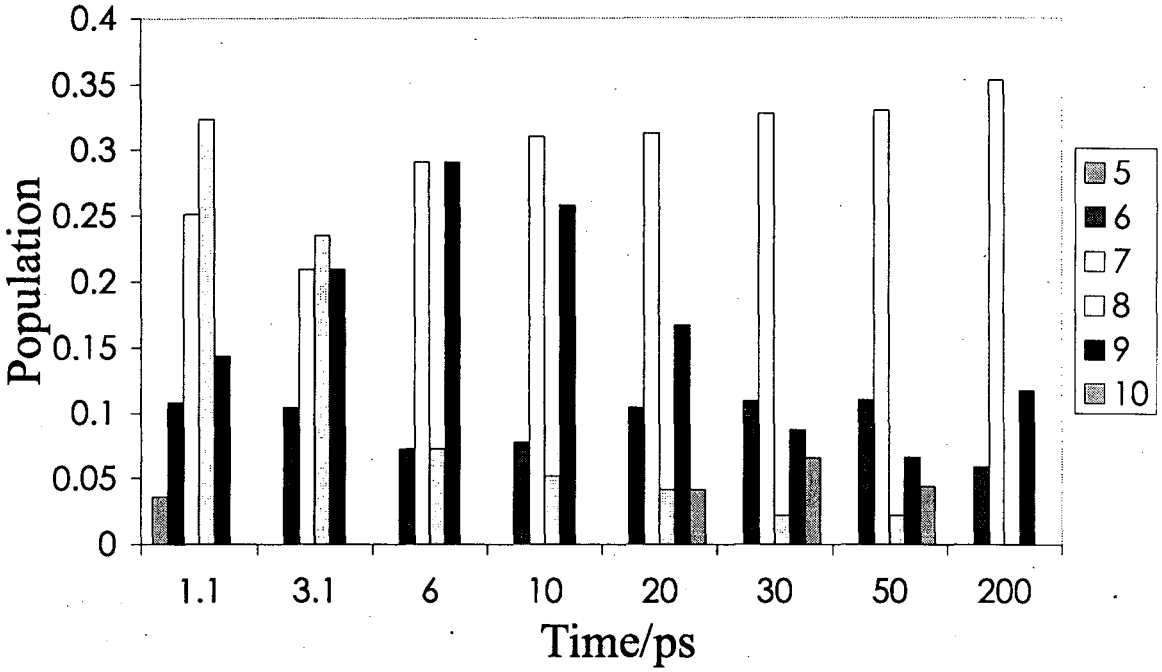


Fig. 11. Population of individual $I^-(CO_2)_n$ contributions to the I^- signal in $I_2^-(CO_2)_{12}$ simulated spectra.

Looking at the time-resolved changes in the other simulation parameters, it is observed that P_X and <v> (Figs. 7a-b) are again inversely correlated, as observed in the smaller clusters. While the $\tilde{X}$ state appears earlier (1.1 ps) than in the smaller clusters, the growth of P_X is actually slower than in $I_2^-(CO_2)_9$, and the initial value of <v> (11.9) is essentially the same as those found in the smaller clusters, probably due to the same averaging effects of a population transfer-limited relaxation rate. <v> does decrease about twice as fast as in $I_2^-(CO_2)_9$, however, so population transfer alone is not

responsible for the relaxation rate. There is no change in $\langle n_X \rangle$ in the simulated spectra (Fig. 8a); however, at 200 ps it is larger than the photofragmentation average of 5.7, and is a larger discrepancy (1.3) than found in the smaller clusters. This increase may reflect the greater energy storage capacity of a larger CO_2 cage, forestalling evaporation.

5.3. $I_2^-(CO_2)_{14}$ and $I_2^-(CO_2)_{16}$

In $I_2^-(CO_2)_{14}$ and $I_2^-(CO_2)_{16}$ clusters, feature E appears at ~500 fs, much earlier than in the smaller clusters, and also undergoes much larger changes in shape, with a distinctly peaked appearance at early time delays (~1 ps), becoming more sloped and moving toward lower eKE as time progresses. Feature F is also visible in both sets of spectra as an intense peak at very low eKE which is present at early times. These features are characteristic of a solvated, highly vibrationally excited I_2^- $\tilde{X}$ state. The presence of solvated I^-, both as a dissociated product and as SS I_2^-, is also observed, though the amount of I^- at long time delays is somewhat ambiguous as only a single broad feature G remains at 200 ps in both clusters. Since little I^- fragments were observed by Vorsa *et al.*,[4,5] the domination of the spectra by these I_2^- $\tilde{X}$ state features is not surprising. We begin by briefly considering the short-time (< ~1 ps) dynamics of solvated I^-, then focus on the simulations.

Since these two clusters possess fairly symmetric solvent configurations, there is little difference between the anomalous and normal estimates for $\langle n_{I^-} \rangle$ (Table 2), precluding definitive assignment of the charge-switching state responsible for the B features. However, there is no reason to suspect that the system is not on the $\tilde{A}'$ state. In fact, it is seen that $\langle n_{I^-} \rangle$ for the B feature is significantly smaller than either estimate, which was also the case in $I_2^-(CO_2)_4$. Rather than being due to rapid separation of I^- from

the cluster, since the I atoms are virtually surrounded by solvent molecules, the discrepancy is probably caused by rapid loss of CO_2 molecules in the "capping" positions (along the I_2^- axis) of the cluster, which Parson and coworkers[34] predict to collide with the I and I^- fragments immediately after photoexcitation. The 0.4-0.5 molecule increase in $\langle n_{I^-} \rangle$ by ~700 fs (feature D) indicates a small increase in solvation. It has been established from earlier discussion that solvent rearrangement on the $\tilde{A}'$ state occurs on the ~500 fs timescale. Presumably, the rapid appearance of solvated I_2^- $\tilde{X}$ (~500 fs) implies that production of I^- on the $\tilde{X}$ / $\tilde{A}$ states occurs on a similar timescale. Thus, the increase in $\langle n_{I^-} \rangle$ producing feature D is probably due to a combination of $\tilde{A}'$ state solvent rearrangement, and transitions to the $\tilde{X}$ / $\tilde{A}$ states which switches the electron to the (slightly) more solvated I atom, followed by solvent rearrangement.

In $I_2^-(CO_2)_{14}$, spectra at three time delays are shown in Figs. 12a-c: 1.3, 5.0 and 200 ps. The 1.3 ps spectrum (P_X = 0.62, <v> = 23.9, $\langle n_X \rangle$ = 10.0, $\langle n_{I^-} \rangle$ = 7.8) exhibits a solvated I_2^- $\tilde{X}$ state spectrum which is very extended, due to the large value of <v>: feature E is accounted for by the $\tilde{X} \leftarrow \tilde{X}$ ITP transition at high eKE, while F is represented at very low eKE by a combination of $\tilde{X} \leftarrow \tilde{X}$ OTP and $I_2^* \leftarrow \tilde{X}$ transitions. Solvated I^- contributes the bulk of the intensity to D. In the 5.0 ps spectrum (P_X = 0.51, <v> = 4.4, $\langle n_X \rangle$ = 9.0, $\langle n_{I^-} \rangle$ = 8.3), <v> is much lower, generating a more compact $\tilde{X} \leftarrow \tilde{X}$ feature which accounts for feature E at much lower eKE, and part of D; the $\tilde{A}' / \tilde{A} \leftarrow \tilde{X}$ transition accounts for F. The solvated I^- contribution peaks at ~300 meV, in between D and F, but represents about 2/3 of the intensity of D. The 200 ps spectrum (P_X = 0.62, <v> = 0.8, $\langle n_X \rangle$ = 9.0, $\langle n_{I^-} \rangle$ = 7.7) consists of a single broad feature G, which is accounted for roughly equally by solvated I_2^- $\tilde{X}$, occupying the high

eKE side of the feature, and solvated I⁻, occupying the low eKE side. The $\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$ transition is also present in the vicinity of F.

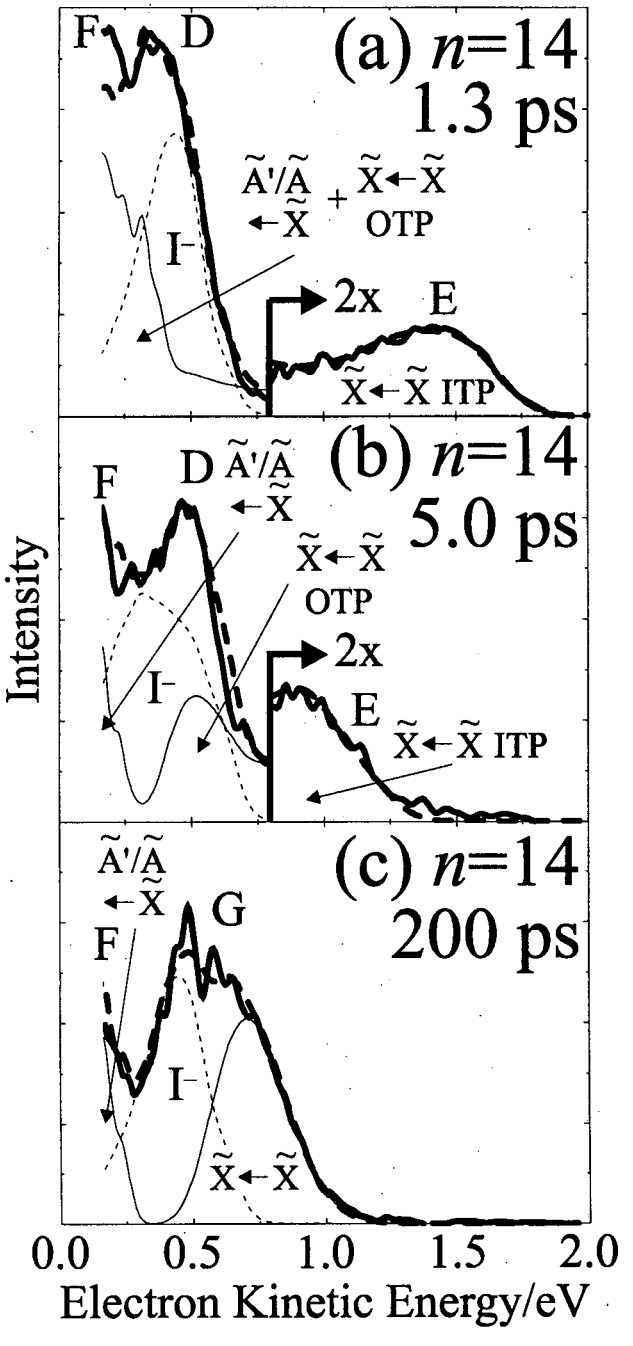


Fig. 12. Simulated $I_2^-(CO_2)_{14}$ spectra at (a) 1.3 ps, (b) 5.0 ps, (c) 200 ps. Experimental (thick solid line) and simulated (thick dotted line) data are shown. Thin lines indicate simulated I⁻ and I_2^- $\tilde{X}$ -based CO_2 fragment contributions.

At 200 ps, $\langle n_X \rangle$ (9.0) is larger than the photofragmentation average (7.6), by about the same amount as for $I_2^-(CO_2)_{12}$, indicating excess energy stored in CO_2 modes which has not yet found its way into evaporation. Although there is significant leeway in the choice of parameters for simulating the 200 ps spectrum, the breadth of feature G in comparison to the $I_2^-(CO_2)_{16}$ spectrum suggests a significant amount of solvated I^- is present, comparable to earlier time delays.

Like the smaller clusters, P_X at 200 ps (0.62) is much smaller than observed in the photofragmentation experiments (0.95), and like $I_2^-(CO_2)_{12}$, $\langle n_{I^-} \rangle$ is also significantly smaller (7.7 *vs.* 9.6), indicating the presence of SS I_2^-. No double-peaked I^- structure is observed, however, as it appears that the dissociated I^- signal is so small that it has little influence on the simulation; with a photofragmentation population of only 0.05, this is not surprising. There is almost no change in the distribution of $I^-(CO_2)_n$ populations *vs.* time delay, other than a slight narrowing after 5.0 ps.

As shown in Fig. 7a, P_X rises much faster than in $I_2^-(CO_2)_{12}$, essentially reaching a plateau at 1.3 ps (0.62). Over this time interval, $\langle v \rangle$ (Fig. 7b) appears to stay constant at 23.9, much larger than in the smaller clusters. Together with the early plateauing of P_X, this higher initial value of $\langle v \rangle$ indicates that the average vibrational level is becoming less controlled by the rate of infusion of population onto the $\tilde{X}$ state, so that all clusters now enter the $\tilde{X}$ state by 1.3 ps with substantial vibrational excitation. There is also a relatively narrow distribution of vibrational levels ($\sigma_v \approx 0.3\langle v \rangle$) compared with smaller clusters, which further illustrates this distinction. Between 1.3 and 20 ps, $\langle v \rangle$ falls to a very low value (1.3), then decreases slightly more (to 0.8) by 200 ps. There is some decrease to $\langle n_X \rangle$ with time (Fig. 8a), falling from 10.0 at 1.3 ps to 9.0 at 5.0 ps, but it is

constant after this time delay. Still, this decrease was not observed in the smaller clusters, and reflects a more concerted vibrational relaxation process, where an initial, rapid evaporation of solvent molecules discarding most of the vibrational energy of the cluster is becoming visible. $\langle n_{I^-} \rangle$ (Fig. 8b) undergoes small changes (±0.7), some of which may be model-dependent, but remains essentially constant near 7.7.

For $I_2^-(CO_2)_{16}$, the following simulated spectra are shown in Figs. 13a-d: 1.0 ps, 2.2 ps, 4.0 ps and 200 ps, respectively, which illustrate the large changes to all features in the spectra. At 1.0 ps ($P_X = 0.68$, $\langle v \rangle = 38.8$, $\langle n_X \rangle = 14.5$, $\langle n_{I^-} \rangle = 8.5$), solvated I_2^- $\tilde{X}$ is responsible for features E ($\tilde{X} \leftarrow \tilde{X}$ ITP) and F ($\tilde{X} \leftarrow \tilde{X}$ OTP), while solvated I^- accounts for D. Note the very intense F feature dominating the spectrum, due to the much larger value of $\langle v \rangle$, which was absent in $I_2^-(CO_2)_{14}$ near this time delay. By 2.2 ps (P_X = 0.85, $\langle v \rangle = 20.9$, $\langle n_X \rangle = 13.0$, $\langle n_{I^-} \rangle = 8.7$), the general shape of the spectrum is unchanged, but E has moved to much lower eKE, and the intensity of F is diminished, due to the considerable $\tilde{X}$ state relaxation and loss of CO_2 molecules compared with 1.0 ps. The $\tilde{X}$ state population has increased significantly as well. The solvated I^- contribution, still accounting for D, is of lower intensity but $\langle n_{I^-} \rangle$ is almost identical. The 4.0 ps spectrum ($P_X = 0.89$, $\langle v \rangle = 6.4$, $\langle n_X \rangle = 12.0$, $\langle n_{I^-} \rangle = 8.5$) could be simulated exclusively by the I_2^- $\tilde{X}$ state, but there is better reproduction of the low eKE region if a small amount of I^- is included in the vicinity of F. The $\tilde{X} \leftarrow \tilde{X}$ ITP transition is responsible for feature E, while the $\tilde{X} \leftarrow \tilde{X}$ OTP transition now dominates feature F. Unlike $I_2^-(CO_2)_{14}$, the $\tilde{A}'/\tilde{A} \leftarrow \tilde{X}$ transitions occur at an eKE too low to observe. The 200 ps spectrum ($P_X = 0.86$, $\langle v \rangle = 1.4$, $\langle n_X \rangle = 11.5$, $\langle n_{I^-} \rangle = 8.9$) reveals further vibrational relaxation to the $\tilde{X}$ state, and whether any I^- is present depends primarily on

the choice of $\langle n_X \rangle$; over the range $\langle n_X \rangle$ = 10.5-12.0, P_X varies from 0.70-1.00. Here we choose a median value, which is most consistent with simulations at earlier time delays. The $\tilde{X} \leftarrow \tilde{X}$ transition accounts for the majority of G, with the solvated I⁻ signal making a small contribution to the low eKE side.

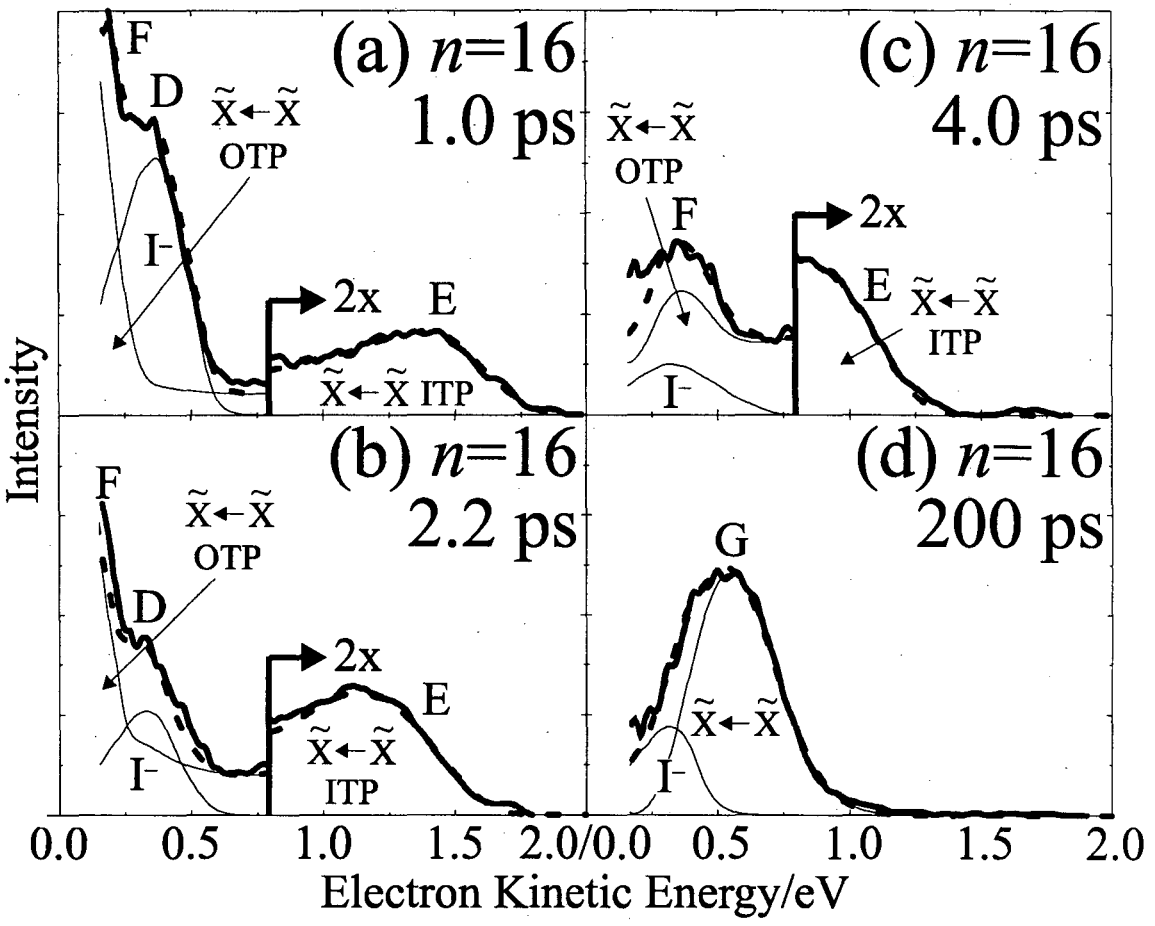


Fig. 13. Simulated $I_2^-(CO_2)_{16}$ spectra at (a) 1.0 ps, (b) 2.2 ps, (c) 4.0 ps, (d) 200 ps. Experimental (thick solid line) and simulated (thick dotted line) data are shown. Thin lines indicate simulated I⁻- and I_2^- $\tilde{X}$ -based CO_2 fragment contributions.

Although spectral evidence is not conclusive, as it was in smaller clusters, it appears that the I⁻ contribution persists through 200 ps. This structure is presumably SS I_2^-, because I⁻ products were not observed in the photofragmentation experiments. The value of $\langle n_I^- \rangle$ (8.9) is also comparable to $I_2^-(CO_2)_{14}$, indicating only about half of the CO_2

molecules surround the I^-, consistent with the SS I_2^- structure. Varying the population of I^- in the 200 ps spectrum affects $\langle n_{I^-} \rangle$ very little.

P_X rises faster than in any other cluster, reaching a plateau of ~0.9 by 2.2 ps (Fig. 7a). Over this time period, <v> drops from an initial value of ~39 to 20.9, and relaxes almost completely by 10 ps (Fig. 7b). The relaxation rate appears slightly slower than in $I_2^-(CO_2)_{14}$ before 2.9 ps, but surpasses it at later time delays. As the initial value of <v> is much larger than in any other cluster, it appears that population is building up in the $\tilde{X}$ state even more rapidly than in $I_2^-(CO_2)_{14}$, after which this large fraction of clusters relaxes in tandem. The width of the distribution is again fairly narrow initially ($\sigma_v \approx 0.3\langle v \rangle$), illustrating the more correlated nature of the relaxation process. As seen in Fig. 8a, $\langle n_X \rangle$ undergoes a large change, from 14.5 at 1.0 ps to 11.5 by 10 ps, indicating simultaneous evaporation of CO_2 molecules as energy leaks from I_2^- into the CO_2 cluster modes. The 200 ps value of $\langle n_X \rangle$ is larger than the photofragmentation average (9.7), by approximately the same amount as in $I_2^-(CO_2)_{12}$ and $I_2^-(CO_2)_{14}$, reflecting the energy residing in the cluster which will eventually dissipate through CO_2 evaporation. $\langle n_{I^-} \rangle$ undergoes little variation (±0.2) with time, after its initial increase through 500 fs (Fig. 8b).

5.4. Trends across cluster size, and comparisons with other studies

The most significant trend across cluster size is the increasingly rapid appearance and subsequent vibrational relaxation of the I_2^- $\tilde{X}$ state. Using the time of initial appearance of feature E ($\tilde{X} \leftarrow \tilde{X}$ transition) as a basis of comparison (see Fig. 7a), a monotonic decrease with cluster size is observed, from ~10 ps in $I_2^-(CO_2)_6$ to ~500 fs in

$I_2^-(CO_2)_{14\text{-}16}$. This order-of-magnitude change is due to the increasing perturbation on the I_2^- electronic states by larger numbers of CO_2 molecules, facilitating nonadiabatic electronic transitions to the $\tilde{X}$ state. The earlier appearance of $\tilde{X}$ state population in larger clusters is also accompanied by initially larger vibrational excitation with a smaller range of levels, indicating the more concerted appearance of I_2^- on the $\tilde{X}$ state and subsequent vibrational relaxation. In smaller clusters, where the rate of population transfer onto the $\tilde{X}$ state was slower than the vibrational relaxation rate of individual cluster molecules, a much lower maximum vibrational level is observed, with a wider distribution of levels.

Although not directly comparable to our determinations of P_X and <v>, the absorption recovery experiments of Vorsa *et al.* confirm the general decrease in time required to reach a low vibrational state, with exponential time constants ranging from 24 ps for $I_2^-(CO_2)_6$ to 1.3 ps for $I_2^-(CO_2)_{16}$.[10] These time constants correspond approximately to the time at which <v> crosses ~7 in Fig. 7b.

While the Parson group model reports a decrease in the initial appearance of the $\tilde{X}$ state with cluster size for n = 6 through 12 (P_X = 0.20 at from ~3.1 to ~1.6 ps),[34] a large fraction of trajectories become trapped in the SS I_2^- configuration on the $\tilde{A}$ state, particularly for $I_2^-(CO_2)_{16}$, so that growth of the $\tilde{X}$ state and vibrational relaxation appear significantly slower than observed experimentally. However, once a trajectory does enter the $\tilde{X}$ state, vibrational relaxation is very rapid in all clusters (1-2 ps),[22] comparable to the rates observed in $I_2^-(CO_2)_{14\text{-}16}$ clusters where relaxation does not appear significantly limited by the electronic transition rate.

The Margulis *et al.* study observes for $I_2^-(CO_2)_8$ recombination in anywhere from 3-27 ps, depending on trajectory, with fast vibrational relaxation (~3 ps). These results are about on par with the FPES data for $I_2^-(CO_2)_9$. However, it should be pointed out this cluster was simulated only at 720 nm, which deposits enough energy in the $I_2^-(CO_2)_8$ cluster to evaporate all CO_2 molecules, whereas ~3 CO_2 molecules remain for $I_2^-(CO_2)_9$ at 780 nm. For $I_2^-(CO_2)_{16}$, the simulations fare little better than the Parson results, as exclusively delayed (10-25 ps) recombination is predicted at 790 nm. The 720 nm results do show a minority of trajectories (~20%) which recombine after the first bond extension at ~700 fs, which is much more consistent with the FPES observations. The paper points toward a possible explanation for this major difference in mechanism: that the initial I_2^- bond distance extends farther at the higher-energy pump excitation (5.5 Å) than at the lower-energy (5.0 Å), facilitating electronic coupling to the $\tilde{X}$ state. Perhaps this apparently sensitive parameter also explains the anomalously high $\tilde{A}$ state population observed in the Parson model.

A by-product of efficient energy absorption by CO_2 is longer energy retention, which is exhibited in the trend of increasing discrepancy between $\langle n_X \rangle$ determined from FPES and measured in the photofragmentation experiments. As cluster size increases, $\langle n_X \rangle$ is seen to increase above the photofragmentation average more and more, growing from essentially 0 in $I_2^-(CO_2)_6$, to 1.8 in $I_2^-(CO_2)_{16}$. The Parson group model also revealed a decrease in the solvent evaporation rate as cluster size increases, but this decrease was compounded by the trapping of a much larger fraction of trajectories on the $\tilde{A}$ state than was supported by the FPES data, delaying the appearance of the $\tilde{X}$ state. Thus, no quantitative comparison could be made between this group's predictions and FPES at

particular time delays. Margulis *et al.* offered no information on their solvent evaporation dynamics.

One of the most interesting results from the FPES study is the identification of the solvent-separated I_2^- structure in $I_2^-(CO_2)_{n \geq 9}$ clusters, which was not observed in the photofragmentation experiments. Features arising from SS I_2^- resemble solvated I^-, because the I_2^- potential is expected to lie very near the dissociation energy when I^- and I are far enough apart to allow a CO_2 molecule to intervene (5-7 Å).[18,22,34] By comparison, the long-lived $\tilde{A}$ state structure in $I_2^-(Ar)_n$ clusters, which is not solvent-separated, exists at a considerably shorter internuclear radius (4.6 Å) and possesses a significant well (140 meV),[31] which serves to shift the photoelectron spectrum dramatically. However, in $I_2^-(CO_2)_n$ clusters, the effect of the $\tilde{A}$ state well is much less important, as it is smaller than the binding energy of a single CO_2 molecule to I^- (212 meV).[40] It is feasible that the SS I_2^- structure could be stabilized by this strong solvent interaction on any of the accessible anion states ($\tilde{X}$, $\tilde{A}$ or $\tilde{A}'$), but Faeder *et al.*[22] predict it to exist exclusively on the $\tilde{A}$ state. In their model, SS I_2^- contains more CO_2 molecules than would be indicated by the I^- solvent shift, with some CO_2's surrounding the neutral I. Indeed, SS I_2^- must contain approximately as many solvent molecules as found in the I^- fragments, since the difference in energy between these structures is the binding energy of neutral I to the cluster, which is estimated[40-42] to be much smaller than the I^--CO_2 well depth. Therefore, the main distinguishing factor between SS I_2^- and dissociated I^- is the apparent $\langle n_{I^-} \rangle$. It was shown that SS I_2^- is only clearly identifiable using $\langle n_{I^-} \rangle$ for $I_2^-(CO_2)_{n \geq 12}$, but that in clusters of $I_2^-(CO_2)_{n \geq 9}$, the large I^- population in comparison to the photofragmentation results also pointed strongly toward an additional I^--like structure, e.g. SS I_2^-.

If the SS I_2^- structure is indeed limited to the $\tilde{A}$ state, its prominence in the spectra is not surprising, as the $\tilde{A}'$ state lies closer in energy to the $\tilde{A}$ than the $\tilde{X}$ state at intermediate internuclear distances, making the $\tilde{A} \leftarrow \tilde{A}'$ coupling stronger than that of the $\tilde{X} \leftarrow \tilde{A}'$. As seen above, the $\tilde{A}$ state is predicted by both theoretical groups to play a major role in I_2^- recombination dynamics;[22,23] indeed, perhaps too great a role! However, in clusters where the $\tilde{X}$ state appears in < ~1 ps [$I_2^-(CO_2)_{n \geq 12}$], the question arises of why the $\tilde{A}$ does not rapidly fall onto the $\tilde{X}$ state. The answer seems to be less a matter of electronic coupling, as all states are essentially degenerate at large internuclear distances, and more of physical constraint in the SS I_2^- structure. Thus, regardless of the electronic state, the I^- and I atoms must overcome the resistance of one or more CO_2 molecules blocking their way to recombination. This is apparently feasible on the timescale of the photofragmentation experiments (~5 μs),[5] but not on the 200 ps timescale.

5.5. Comparison with $I_2^-(Ar)_n$ clusters

Comparing the $I_2^-(CO_2)_n$ cluster results to those of $I_2^-(Ar)_{20}$, the most obvious difference is the much faster rate of I_2^- vibrational relaxation in $I_2^-(CO_2)_n$ clusters, despite the fact that the I_2^- $\tilde{X}$ state grows at approximately the same rate for $I_2^-(CO_2)_{n \leq 9}$ (see Fig. 7a). This distinction highlights the existence of the two separate effects involved in I_2^- recombination: the ability to remove energy from the I_2^- bond, and the ability to facilitate an electronic transition to the $\tilde{X}$ state in the first place. It is clear that CO_2 is far more effective than Ar in removing energy, but appears no better than Ar in promoting an electronic transition for $I_2^-(CO_2)_{n \leq 9}$. However, there is an increased long-term ability to

return I_2^- to the $\tilde{X}$ state, exhibited by the lack of any $\tilde{A}$ state products in the photofragmentation studies (~5 μs timescale).[5] It should also be mentioned that clusters as small as $I_2^-(CO_2)_4$ exhibit some I_2^- recombination, which is not observed in $I_2^-(Ar)_n$ clusters until $n = 10$. This last point, however, illustrates the ability of $CO_2$to prevent I_2^- from dissociating (often trapping the system in an SS I_2^- configuration), rather than its efficacy in promoting a speedy transition to the $\tilde{X}$ state.

The increasingly large discrepancy between the FPES and photofragmentation values of $\langle n_X \rangle$ in $I_2^-(CO_2)_n$, which was absent in $I_2^-(Ar)_n$, illustrates another difference between solvent types: the ability to retain energy in the solvent cage. Ar lacks any internal degrees of freedom, and the Ar-Ar interaction is also weaker than CO_2-CO_2, making Ar far less effective than CO_2 in storing energy liberated from I_2^- vibration. Thus, whereas $I_2^-(CO_2)_n$ clusters may store a great deal in internal and collective cluster modes, forestalling evaporation until time delays much longer than 200 ps, $I_2^-(Ar)_n$ clusters must rapidly evaporate Ar atoms to dissipate internal energy, and $I_2^-(Ar)_{20}$ has almost no Ar atoms left by 200 ps.

At early time delays (< ~1 ps), $I_2^-(CO_2)_n$ and $I_2^-(Ar)_n$ clusters exhibit dramatically different solvent motions, with the number of solvent molecules increasing in $I_2^-(CO_2)_n$, while decreasing in $I_2^-(Ar)_n$. The increase in $I_2^-(CO_2)_n$ clusters has been explained by solvent molecules strongly affecting the I^- and I motion via solvent reorganization on the $\tilde{A}'$ state, and/or electronic transitions to the $\tilde{X}$ / $\tilde{A}$ states, whereas the decrease in $I_2^-(Ar)_n$ clusters reflects the much weaker influence of the Ar atoms on the motion of I^- and I, as I^- is able to separate considerably from the cluster before an electronic transition occurs.

Both types of clusters also exhibit long-lived metastable structures, although it is known for $I_2^-(CO_2)_n$ clusters that this state is not stable on the μs timescale of photofragmentation experiments. In $I_2^-(Ar)_n$ clusters, the bond length of I_2^- on the $\tilde{A}$ state is short enough so that a large energy shift is seen in the photoelectron spectra, distinguishing this state spectroscopically from solvated I^-. In $I_2^-(CO_2)_n$ clusters, the much stronger solvation energy of I^--CO_2 enables a solvent-separated structure to form on the $\tilde{A}$ state, keeping the I_2^- bond long enough (estimated at 5-7 Å)[34] so that it is spectroscopically indistinguishable from I^-. Presumably it is the difference in bond length which determines the longevity of this state, because the $\tilde{A}$ and $\tilde{X}$ states lie much closer together at long bond lengths than at the $\tilde{A}$ state equilibrium distance (4.6 Å),[31] facilitating an $\tilde{X} \leftarrow \tilde{A}$ transition on the > 200 ps timescale. Experiments probing $I_2^-(CO_2)_n$ clusters in the ns regime should be conducted to look for evidence of increased $\tilde{X}$ state population.

6. Conclusions

The time-resolved photodissociation dynamics of I_2^- in CO_2 van der Waals clusters have been investigated using FPES for a range of sizes covering the uncaged and fully-caged product limits. In all clusters, solvated I^- is produced on the anomalous charge-switching $\tilde{A}'$ state in ~200 fs, followed by solvent rearrangement on this state occurs by ~500 fs, increasing the number of CO_2 molecules around I^-. Electronic transitions to the normal charge-switching $\tilde{X}$ and/or $\tilde{A}$ states occur by ~800-1.1 ps, further increasing the number of CO_2 molecules by returning the electron to the more solvated I atom. In $I_2^-(CO_2)_4$, the reaction is essentially over at 800 fs.

In $I_2^-(CO_2)_{n \geq 6}$ clusters, the transition to the $\tilde{X} / \tilde{A}$ state induces I_2^- recombination, the fraction of which increases with cluster size. For $I_2^-(CO_2)_{n \leq 12}$, recombination is the rate-limiting step in vibrational relaxation, resulting in a broad distribution of vibrational levels which decreases from $\langle v \rangle \approx 9$-12 initially to $\langle v \rangle \approx 1$ by 200 ps; in $I_2^-(CO_2)_6$, the final $\langle v \rangle$ is much higher, ~5, due to the small number of CO_2 molecules present (0-1). In $I_2^-(CO_2)_{14\text{-}16}$, recombination occurs much faster, and higher initial values of $\langle v \rangle$ are observed (up to ~40), with narrower distributions. The overall rate of vibrational relaxation increases dramatically from $I_2^-(CO_2)_6$ to $I_2^-(CO_2)_{16}$.

The numbers of solvent molecules around I^- and I_2^- products at 200 ps are larger than those observed in the photofragmentation experiments, implying that evaporation of CO_2 from the cluster occurs on a much longer timescale. This is consistent with the assumption that the CO_2 cage stores a considerable amount of energy in vibrational and/or cluster modes after removing it from I^- kinetic energy or I_2^- vibration. The discrepancy increases with cluster size, illustrating the increasing energy "storage capacity" of larger clusters.

In clusters of $I_2^-(CO_2)_{n \geq 9}$, a solvent-separated I_2^- structure is observed at long time delays, which appears spectroscopically as solvated I^-. This structure is needed to explain the high intensity of I^- features observed in these spectra, and the small number of CO_2 molecules surrounding I^-. While the lifetime of this metastable state appears to be > 200 ps (the longest time delays measured), it is not present in the photofragmentation experiments, which have a ~5 μs duration. The strong I^--CO_2 bond is thought to be responsible for the stability of the solvent-separated structure.

7. Acknowledgments

The authors would like to thank James Faeder, Nicole Delaney and Professor Robert Parson for many helpful discussions. Funds supplied by the National Science Foundation under Grant No. CHE-9710243, and the Defense University Research Instrumentation Program under Grant No. F49620-95-1-0078, are gratefully acknowledged.

8. References

[1] M. L. Alexander, N. E. Levinger, M. A. Johnson, D. Ray, and W. C. Lineberger, J. Chem. Phys. **88,** 6200 (1988).

[2] S. Nandi, A. Sanov, N. Delaney, J. Faeder, R. Parson, and W. C. Lineberger, J. Phys. Chem. A **102,** 8827 (1998).

[3] A. Sanov, S. Nandi, and W. C. Lineberger, J. Chem. Phys. **108,** 5155 (1998).

[4] V. Vorsa, Ph.D. Thesis, University of Colorado, Boulder (1996).

[5] V. Vorsa, P. J. Campagnola, S. Nandi, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **105,** 2298 (1996).

[6] D. Ray, N. E. Levinger, J. M. Papanikolas, and W. C. Lineberger, J. Chem. Phys. **91,** 6533 (1989).

[7] J. M. Papanikolas, J. R. Gord, N. E. Levinger, D. Ray, V. Vorsa, and W. C. Lineberger, J. Phys. Chem. **95,** 8028 (1991).

[8] J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, J. R. Gord, and W. C. Lineberger, J. Chem. Phys. **97,** 7002 (1992).

[9] J. M. Papanikolas, V. Vorsa, M. E. Nadal, P. J. Campagnola, H. K. Buchenau, and W. C. Lineberger, J. Chem. Phys. **99,** 8733 (1993).

[10] V. Vorsa, S. Nandi, P. J. Campagnola, M. Larsson, and W. C. Lineberger, J. Chem. Phys. **106,** 1402 (1997).

[11] L. Perera and F. G. Amar, J. Chem. Phys. **90,** 7354 (1989).

[12] F. G. Amar and L. Perera, Z. Phys. D. **20,** 173 (1991).

[13] P. E. Maslen, J. M. Papanikolas, J. Faeder, R. Parson, and S. V. ONeil, J. Chem. Phys. **101,** 5731 (1994).

[14] J. M. Papanikolas, P. E. Maslen, and R. Parson, J. Chem. Phys. **102,** 2452 (1995).

[15] P. E. Maslen, J. Faeder, and R. Parson, Chem. Phys. Lett. **263,** 63 (1996).

[16] V. S. Batista and D. F. Coker, J. Chem. Phys. **106,** 7102 (1997).

[17] J. Faeder, N. Delaney, P. E. Maslen, and R. Parson, Chem. Phys. Lett. **270,** 196 (1997).

[18] N. Delaney, J. Faeder, P. E. Maslen, and R. Parson, J. Phys. Chem. A **101** (1997).

[19] B. M. Ladanyi and R. Parson, J. Chem. Phys. **107,** 9326 (1997).

[20] J. Faeder and R. Parson, J. Chem. Phys. **108,** 3909 (1998).

[21] P. E. Maslen, J. Faeder, and R. Parson, Mol. Phys. **94,** 693 (1998).

[22] J. Faeder, N. Delaney, P. E. Maslen, and R. Parson, Chem. Phys. **239,** 525 (1998).

[23] C. J. Margulis and D. F. Coker, J. Chem. Phys., in press (1999).

[24] A. E. Johnson, N. E. Levinger, and P. F. Barbara, J. Phys. Chem. **96,** 7841 (1992).

[25] D. A. V. Kliner, J. C. Alfano, and P. F. Barbara, J. Chem. Phys. **98,** 5375 (1993).

[26] J. C. Alfano, Y. Kimura, P. K. Walhout, and P. F. Barbara, Chem. Phys. **175,** 147 (1993).

[27] I. Benjamin, P. F. Barbara, B. J. Gertner, and J. T. Hynes, J. Phys. Chem. **99,** 7557 (1995).

[28] P. K. Walhout, J. C. Alfano, K. A. M. Thakur, and P. F. Barbara, J. Phys. Chem. **99,** 7568 (1995).

[29] B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Science **276,** 1675 (1997).

[30] B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Faraday Discuss. **108,** 101 (1997).

[31] B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, J. Chem. Phys., to be submitted .

[32] B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Chem. Phys. Lett. **258,** 523 (1996).

[33] K. Asmis, T. Taylor, and D. M. Neumark, J. Chem. Phys. **109,** 4389 (1998).

[34] N. Delaney, J. Faeder, and R. Parson, private communication .

[35] W. C. Wiley and I. H. McLaren, Rev. Sci. Instrum. **26,** 1150 (1955).

[36] O. Cheshnovsky, S. H. Yang, C. L. Pettiette, M. J. Craycraft, and R. E. Smalley, Rev. Sci. Instrum. **58,** 2131 (1987).

[37] L.-S. Wang, H.-S. Cheng, and J. Fan, J. Chem. Phys. **102,** 9480 (1995).

[38] H. Gomez and D. M. Neumark, in progress .

[39] M. T. Zanni, V. S. Batista, B. J. Greenblatt, W. H. Miller, and D. M. Neumark, J. Chem. Phys. **110,** 3748 (1999).

[40] Y. Zhao, C. C. Arnold, and D. M. Neumark, J. Chem. Phys. Faraday Trans. **89,** 1449 (1993).

[41] D. W. Arnold, S. E. Bradforth, E. H. Kim, and D. M. Neumark, J. Chem. Phys. **102,** 3510 (1995).

[42] D. W. Arnold, S. E. Bradforth, E. H. Kim, and D. M. Neumark, J. Chem. Phys. **102,** 3493 (1995).

Appendix 1. Data Acquisition Program (fpes)

1. Program overview

The primary purpose of the data acquisition program (fpes) is to record photoelectron spectra, mass spectra and auto/cross-correlation spectra. However, because of the need to record photoelectron spectra at many time delays, and because the program is also used to analyze and manipulate these data, a significant amount of extra functionality has been added to aid in these tasks. For instance, several spectra, not necessarily of the same type, can be held in memory and displayed simultaneously; the screen format can be a single window or several side-by-side windows; zoom-in capabilities are supported; peak heights, positions, widths and integrated areas can be measured automatically; smoothing and summing of data is possible; data can be displayed with alternate units (for instance, a photoelectron spectrum may be displayed either on a time or energy axis) and calibration for use of these formats is possible. The user interface is also fairly sophisticated, replacing the "one key, one function" approach of earlier programs in the group with a command word format, allowing for infinite expansion.

At its largest level, the program operates in a loop, alternating between two procedures, com_rd and com_ex (see fpes.pas). com_rd itself operates in a loop, updating active processes related to data acquisition (scan procedure), and then checking for a keypress. If a key has been entered, com_rd either displays it on the screen while simultaneously adding it to a command line buffer (tx_wr_ch procedure), or acts on the key immediately. Pressing the enter key ("carriage return," or CR for short)

ends the loop, causing the typed command line to be processed or "parsed" for content (com_parse procedure). The processed command is then executed by com_ex.

Commands are designed to use as many modular procedures as possible. For instance, commands which modify the graphics display do not write to the screen directly; they only change the controlling variables (sc or wv). The visual change is caused by calling an updating procedure, update or updateall, upon exiting. The syntax for commands is detailed in the help.par file (readable via the help command)

2. Selected variables

There are too many variables in fpes for an exhaustive survey, but selected structured and simple variables will be discussed here. The documentation contained within the source code should be sufficient to at least clarify the intended use of other variables used by the program.

Several of these structures (com, sc, wv) contain variables with similar functions: ls, a "list" or array of a kind of information (such as individual spectra); cur, the index of the current element being manipulated; and num, the total number of such elements in memory. In addition, both sc and wv have a sel ("select") variable which many commands use when a list has been specified as part of the input line. Those screens or waves in the list will have their sel variables set (= 1) to select them for action; the command then acts on each selected item, after which the sel variables are cleared (= 0).

To refer to an element of a structure in Pascal, a period (.) is used between the structure name and its element. For instance, to access the num variable within the sc

structure, the syntax is `sc.num`. It can be extended for more complex references, e.g. `sc.ls[1].mode.gr.xh.u[2]`.

2.1. Background subtraction: _bs

Variable in _bs	Function
bg	Background wave and mode
fg	Foreground wave
dis	Display mode simulating background-subtraction waves (alternating scan mode)
sts	Shot-to-shot mode
sts_blank, sts_blank2	Point range for blanking background wave
sts_ch	Oscilloscope channel of background wave (shot-to-shot mode)
sts_fac	Scaling factor (shot-to-shot mode)
sts_tog	Toggle mode (shot-to-shot mode)
sts_vert	Vertical scale of background wave on oscilloscope (shot-to-shot mode)

The variables pertaining to background subtraction have been collected in `_bs` (the underscore "_" is to distinguish it from `bs`, a substructure within `wv`). Background subtraction is used when collecting electron spectra, in order to compensate for changing signal levels due to the ions and/or laser, and concerns the collection of a "background" spectrum, generally one obtained in the absence of the pump laser. There are two kinds of background subtraction: "alternating scan" and "shot-to-shot." Alternating scan background subtraction indicates alternation between various positive time delays and a

fixed, negative time delay, such that the effect of the pump laser is absent. Shot-to-shot background subtraction requires the participation of the New Focus optical chopper (see Experimental apparatus chapter) to block the pump laser every other laser shot; photoelectron spectra are either added or subtracted from the accumulating average on the Stanford Research Systems Multichannel Scalar (MCS), depending on whether or not the pump beam is present. A separate, probe-beam only ("background") spectrum is collected at the same time using the Tektronix digitizing oscilliscope.

In order to use either background subtraction mode, the background wave must be specified by `bg`; if it is `0`, no background subtraction is performed. The type of background subtraction is indicated by `sts` ("shot-to-shot"), being equal to `0` for alternating scan background subtraction, and `1` for shot-to-shot background subtraction. Note that, in addition, a participating wave must have its `par.dt.ele.bs.mode` variable set (= `1`); see `wv` variable.

The `dis` variable is useful only in alternating scan background subtraction mode, and enables (when equal to `1`) a display mode whereby participating waves appear with the background wave subtracted from it automatically, though the unsubtracted, raw data is actually stored in the wave.

`sts_tog` (accessed with the `sts tog` command) indicates a choice of "toggle" modes useful only in shot-to-shot background subtraction, where the action taken by the MCS during a background acquisition can be modified. It is normally set (= `1`), indicating that background spectra are subtracted from the accumulating data. When cleared to `0`, it inhibits any action whatsoever, which can be useful in certain circumstances, e.g. when a pump + probe signal is desired to be recorded. Note that the

BNC cable connection from the New Focus chopper controller to the back of the MCS must be changed accordingly (for `sts_tog = 1`, use "toggle" input ; for `sts_tog = 0`, use "inhibit" input).

The other variables in `_bs` are relatively minor.

2.2. Command line: com

Variable in com	Function
`ls[]`	Array of words comprising command
`cur`	Current word being processed
`num`	Number of words in command
`old`	Last string typed (for `!!` command)
`sv`	Current string typed
`tx`	Text screen variables:
`.bdy[]`	Bounding coordinates of command screen
`.buf^.`	Character buffer pointer
`.col^`	Color array pointer (one per line)
`.cur`	Current cursor coordinates
`.num`	Character dimensions of screen
`ystart`	Command screen current starting line

The `com` structure is used to record command input. It has a two-stage "life," first used to record characters as the command line is input and edited (`tx` structure), then used to store the processed "words" (characters separated by spaces) comprising the

command (`ls` array). The remaining variables are either used by text input procedures (`old`, `sv` and `ystart`) or command execution procedures (`cur` and `num`). `cur` is an important variable, used by numerous procedures, and indicates the current position in the array where a procedure has "read" to.

`tx`, a `tx_type` structure, was originally designed to be used by both `com` and `sc` (screens structure), as `fpes` was planned to allow each screen to operate in either a text or graphics mode. The text capability was not built, however, so `tx_type` applies only to the `com.tx` structure. The main variable is `buf`, a storage area containing the characters visible on screen, which is needed for editing and also for when the graphics screen is redrawn. `col` contains a color code for each line of text. Note that both of these variables are pointers, which allow their sizes to be altered within the program (in practice, however, this feature is unnecessary, since the size of the command screen can be fixed before compilation). The other variables are of relatively minor importance.

The `ls` array contains simple strings which `com_parse` creates as it reads through the command line. For more information on this process, see the `com_parse` procedure.

2.3. Screens: `sc`

Variable in `sc`	Function
`bdy`	Overall screen bounding coordinates
`cur`	Current screen
`ls[]`	Array of screen variables:
`.sel`	Selection flag
`.ti`	Screen title variables

.mode.gr	Screen mode variables:
.bdy	Bounding coordinates of screen
.cursor*name*	Cursor variables
.plotarea	Bounding coordinates of plotting area
.maxxnums, .maxynums, .u*name*, .v*name*, .x*name*, .y*name*	Axis variables
.xh	Crosshairs variables:
.bitmap	Storage area for graphics under crosshairs
.mode	Mode (on or off)
.u[], .v[], .x[], .y[]	Coordinates of crosshairs
.which	Active crosshairs
mode	Screen mode
num	Number of active screens
sel	Overall selection indicator flag

One of the most poweful features of fpes is its multiple or "split" screen display ability. It is accomplished through the sc structure, which contains, in addition to some general variables, the screen list array ls, each element of which has all the variables needed to display a complete graphics screen.

The mode variable enables alternation between text (= sc_mode_TX) and graphics (= sc_mode_GR) formats. The text format is used for wave editing (ed command), and a few other isolated commands, such as calibration (cal command).

While using text format, modifications made to a graphics screen (for instance, by data acquisition procedures) are normally suppressed, though when the display reverts to graphics format, the changes are implemented, since the controlling variables were modified, not the screen itself. However, it is sometimes necessary to revert back to the graphics format while the program is using text format, such as when an error occurs in the middle of data acquisition. Therefore, a third mode, `sc_mode_TX_OVR`, is available to allow for this "override" possibility (for details of conditions, see the `tx_wr` procedure).

Actual switching between modes is a bit convoluted. To enter text mode, the procedure `TextMode` is called, which automatically changes `mode` to `sc_mode_TX`. To enter `GraphicsMode`, the `UpdateAll` procedure must be used, which changes `mode` back to `sc_mode_GR`. `mode` may be switched between `sc_mode_TX` and `sc_mode_TX_OVR` at will, since there is no immediate change to the screen.

The other variables in the main `sc` structure are minor.

Within `ls` are a few minor variables, and `mode`, the main variable. `mode` is a `case` variable, which means it can refer to more than one kind of variable, depending on its value. As described in the section on `com`, `fpes` was originally planned to allow each screen to operate in either a text or graphics mode. The text capability was not built, however, so only the `gr_type` ("graphics") structure is used, and `mode` is always set to `sc_mode_GR`. Information is accessed with the syntax `mode.gr` (for discussion of `case` variable usage, see the `dt` variable in the `wv` section).

Most of the variables within `gr` are used to manage the unit translation and axis labeling features of the screen; these variables all start with the letter `u`, `v`, `x` or `y` (plus

maxxnums and maxynums). The remaining variables deal with the cursor position (cursor*name* variables), with the crosshairs (xh structure) or with screen size (bdy and plotarea).

The crosshairs, when active (xh.mode = 1), are composed of three horizontal and three vertical lines, one of which is always midway between the other two for each direction. The positions of these lines are stored both as screen coordinates (x and y arrays) and as unit coordinates (u and v arrays). One line of each direction is "active," meaning it is the one which moves from keyboard input; this is indicated by which. Finally, because the crosshairs must quickly move over a screen which may contain complicated graphics, the entire screen cannot be redrawn every step. Therefore, when the crosshairs are drawn, a copy of those sections of the screen underneath them are recorded in the bitmap structure, and they are restored when the crosshairs are erased, using built-in graphics procedures.

2.4. "Waves" (spectra): wv

wv is probably the most important variable in the program, since it stores photoelectron spectra, as well as other data. It is also the largest and most complex variable. It is so big that pointers must be used to access it, since it encompasses more than 64 kbytes of memory; fortunately, little additional work is needed to utilize pointers, other than a few initialization routines (see Initialize procedure), and a carat (^) symbol after the ls[] variable.

Variable in wv	Function
`cur`	Current wave
`ls[]^`	Array of wave variable pointers:
`.data[]`	Intensity array
`.tmp[]`	Temporary array (for `redo`)
`.par`	Parameter variables:
`.dt`	Datatype and datatype-dependent variables
`.alert`, `.comment`, `.fn`, `.gen`, `.pt`, `.pt_gl`, `.scan`, `.scan_gl`, `.sh`, `.skip`, `.timeperpt`, `.vstop`, `.yoffset`, `.yscale`	Other variables
`.scan`	Scan variables:
`.accum`	Accumulator for `cor` and `pow` data
`.cycle`	Number of cycles (see `par.skip`)
`.mode`	Scan status
`.shots`	Number of laser shots
`.starttime`	Time of most recent scan
`.steptime`	Time to wait before updating
`.col`, `.datasaved`, `.lines`, `.mass1`, `.on`, `.parsaved`, `.pmin`, `.savemode`, `.screen`, `.sel`, `.time1`, `.vmax`, `.vmin`	Other miscellaneous variables
`Num`	Number of waves

Sel	Overall selection indicator flag
temp[]	Temporary wave used in smoothing

At the top level of wv are a few bookkeeping variables (cur, num, sel), a temporary array (temp), and ls, the wave list array (see figure). Within each element of ls is the information about each wave. It contains the actual data array (data), a temporary data array (tmp), a parameter structure (par), a scan structure (scan), and several other variables. The miscellaneous variables can be grouped into several categories: display features (col, lines, on, screen, sel), save status (datasaved, parsaved, savemode), calibration (mass1, time1), and display mode-specific quantities (pmin, vmin, vmax).

The data and tmp arrays each have a MAXPOINTS (currently 1024) number of elements, sufficient for a photoelectron or mass spectrum, and more than enough for a correlation spectrum. The data array is meant to store an accumulating spectrum, while the tmp array is used to store the most recently acquired spectrum, in order to provide a means of subtracting it from data if there is a problem (see redo command). The tmp array type is integer, which uses half the memory of the real type, an important memory-saving trick, since data must be able to take on noninteger values for arbitrarily adding and scaling waves, but tmp does not require these capabilities.

The par ("parameter")structure contains information displayed when editing a wave (see ed command). The most important variable here is dt, "datatype," which represents the kind of spectrum stored in the wave. There are currently four datatypes defined, represented by constants:

Constant	Actual value	Datatype
dt_COR	1	Correlation
dt_ELE	2	Electron
dt_POW	3	Power
dt_MAS	4	Mass

The actual values are irrelevant, as the constants are always used. The power datatype is seldom used, and caution should be exercised when programming for it, as procedures associated with the power datatype may no longer be fully functional.

Parameters are divided into two groups. Those applicable to all waves, regardless of datatype, are placed in the main `par` structure, and appear on the top half of the editing screen in white. Those specific to a certain datatype are accessed through `dt` using the `case` expression, and appear on the bottom half of the editing screen in green. `case` allows several different variables to be accessed, depending on the value of a governing variable (`dt`). Each possible variable has a different name (`cor`, `ele`, `pow` and `mas`, respectively), which are themselves structures. For instance, if `dt` = `dt_COR`, then access to the correlation structure is specified by `par.dt.cor`. Note that no error checking is performed when accessing a particular `case` variable; it is up to the programmer to know the value of `dt` at all times.

The `scan` structure stores acquisition status information, used by all datatypes but with slightly different functions for each. The most critical variable is `mode`, which indicates the current state of an acquisition sequence; see the `Scan` procedure for detailed information.

2.5. Local variables

Within individual procedures are "local" variables where, for general purposes, the same names are usually used. Here is a partial list of these variables:

Name	Type	Function
`c`	`char`	Keypress, string manipulation
`dummy`	`integer`	"Dummy" (unused) variable, used with the `val` built-in procedure to convert a string to a number
`exitflag`	`boolean`	Flag to signal exit from loop
`f`	`text`	File specifier (for reading/writing files)
`i, j, k, n`	`integer`	Counting, array indices
`r`	`real`	Real-number calculation
`s, s2, etc.`	`bufstring`	String manipulation
`temp`	`bufstring, integer`	Temporary values
`w`	`integer`	Wave index

3. Compiling and execution

The program is written in Borland Pascal 7.0. The easiest way to compile the program is to use the `make` menu command within the compiler, after specifying `fpes.pas` as the "target" file. Alternatively, each unit can be compiled separately (creating *name*`.tpu` object files), and then `fpes.pas` may be compiled; this is what `make` does automatically. The result in either case is an executable file, `fpes.exe`. The only additional complication is the necessity of executing a single DOS command, for serial port initialization, prior to running `fpes.exe`. This awkward step has been alleviated with a DOS batch file, `fs.bat`, which accomplishes these tasks with a single

command, `fs` (from DOS). Note that it is currently not possible to run `fpes` from Windows if data acquisition is required (it is fine simply for data manipulation).

Running the program entails simply typing `fs` from the DOS command line.

4. Program listing

The `fpes` program is divided into a number of files (*name*`.pas`), all but one (`fpes.pas`) of which are Pascal "units," an archaic but necessary organization used to run a program in the DOS environment. Units are limited to 64 kbytes when compiled, and to make matters worse, the total memory of program plus variables cannot exceed 640 kbytes. As `fpes` has grown, the number of waves simultaneously held in memory has had to decrease accordingly. On the bright side, the unit organization of the program has been used advantageously to subdivide logical sections of the program. An overview of the files and their contents are as follows:

`fpes.pas` (not a unit)	Main procedure only (highest level of organization)
`fpescom.pas`	Command processing (`com_`*name*) and command action (`do_`*name*) procedures, organized alphabetically

`fpesai.pas`
`fpesjr.pas`
`fpesst.pas`
`fpesuz.pas`
	All other procedures and functions, organized alphabetically over several units

`fpesvar.pas`	All constants, types and variables, organized into logical sections; within each section, in order as `const`, `type`, and `var`; within each category, alphabetically
`dosshell.pas`	Procedures for limited DOS functionality within the `fpes` program, originally designed to be used by other programs (hence separate unit)
`keys.pas`	Constant definitions for nonprinting characters used by `fpes`
`tpdecl.pas`	Interface structure for the National Instruments GPIB card (executable code contained in `tpib.obj`, not listed here)

The above files are collectively referred to as the "`fpes` program," and version 6 of this program is presented in the files which follow.

In addition, the following files are important enough to list along with the units:

`fs.bat`	DOS batch file to initialize serial port before running `fpes.exe`
`help.par`	Syntax of all commands (viewable from within program using `help` command)
`mass.par`	Mass calculator parameters (example)

4.1. fpes.pas

```
{$M 16384, 0, 327680}
{$N+}
program fpes (input, output);

{ History of modifications (please add to BOTTOM of list!):

  Version 1: Begun 28may94 BJG.

  Style notes for program entities:
  lowercase: for commands, types, and variables;
  CapitalizedWords: for program and unit procedures and functions (not com-
    mands!);
  ALLCAPITALS: for constants.

  Version 6: 1998-6-3-Wed BJG
  NB: Sorry no other notes have been recorded over the years -- this
  situation will be rectified when I write up my thesis.

  Current version has made the following modifications over version 5
  (to be itself documented later):
  PtoU: Quadratic energy formula modified so as to not crash program
    when attempting to smooth in energy space (sm en).  Simply set the
    temp variable to zero when temp < 0, rather than setting entire
    PtoU return to 0 (this is because sqrt(temp) is calculated next).
    This prevents a large discontinuity in the value of points at low
    energy, in the case where temp < 0, and allows smooth to work fine.

  1998-6-8-Mon BJG
  Realized the reason versions 3 and later can't take mass spectra is
  because the TEK initialization code in ScanMAS was deleted!  So I
  put it back in.
}

uses
  {Calib5, }crt, dos, FpesCom, FpesVar, FpesAI, FpesJR, FpesST,
  FpesUZ, graph,
  Keys;

begin
  { Note: Version # is in VERSION variable; see FPESVAR.PAS }
  Initialize; { Initialize all internal variables, check for workspace
    default file on disk (use internal defaults if none). }
{ IntroduceProgram; { Print friendly greeting. }
  UpdateAll; { Draw waves with proper bells & whistles }
  while exitflag = false do
  begin
    com_rd; { Read command line; also process "hot keys." }
    com_ex; { Execute commands. }
  end;
  ws_sv(WS_FN_DF); { Save workspace. }
  TidyUp; { Deallocate memory. }
end.
```

4.2. fpescom.pas

```
unit FpesCom;
{$M $4000,0,0 }

interface

uses
  FpesVar;

procedure com_err;
procedure com_err_wv(w : integer; s : bufstring);
procedure com_ex;
procedure com_parse(s : bufstring);
procedure com_rd;
procedure com_wr(s : string; col : word);
procedure com_wr_db(s : bufstring);
procedure com_wr_wv(w : integer; s : string; col : word);
function com_wr_yn(s : bufstring) : integer;
function com_wr_ynaesc(s : bufstring) : integer;
procedure com_wr_sv;
procedure do_abs;
procedure do_ac_cc(pos : real);
procedure do_ad;
procedure do_add;
procedure do_auto;
procedure do_blank;
procedure do_bg;
procedure do_bg_chg(i : integer);
procedure do_bs;
procedure do_bs_dis;
procedure do_bs_mode(i : integer);
procedure do_cal;
procedure do_cd;
procedure do_cp;
procedure do_cpd;
procedure do_dos;
procedure do_dots;
procedure do_ed;
procedure do_fit;
procedure do_fn;
procedure do_fn_x;
procedure do_fpes;
procedure do_gen;
procedure do_macro;
procedure do_mc;
procedure do_mon;
procedure do_mv;
procedure do_nw;
procedure do_osc;
procedure do_rd;
procedure do_redo;
procedure do_rm;
procedure do_run(blank : integer);
procedure do_sc;
procedure do_sc_ss(w : integer);
procedure do_sech;
procedure do_sh;
procedure do_sm;
procedure do_ss;
procedure do_stop;
```

```
procedure do_sts;
procedure do_sv;
procedure do_t0;
procedure do_ts;
procedure do_ts_dly;
procedure do_ts_tog;
procedure do_tw;
procedure do_vis(v : integer);
procedure do_ws;
procedure do_wv;
procedure do_wv_alert;
procedure do_wv_bg;
procedure do_wv_dly;
procedure do_wv_fn;
procedure do_wv_info(w : integer; s2 : bufstring);
procedure do_wv_lines(l : integer);
procedure do_wv_sc;
procedure do_wv_sh;
procedure do_wv_skip;
procedure do_wv_vstop;
procedure do_x(scr : integer);
procedure do_xh;
procedure do_y(scr : integer);
procedure do_yoff;
procedure do_ysc;

implementation

uses
  crt, dos, FpesAI, FpesJR, FpesST, FpesUZ,
  graph, Keys, DOSShell, TPDecl;

procedure com_err;
{ Print error message with some diagnostics to help user figure out
  what's wrong. }
var
  s : bufstring;
begin
  { Convert current word number to string. }
  str(com.cur, s);
  com_wr('Error in word ' + s + ': "' + com.ls[com.cur] + '"',
    COLORHL);
  wv_sel_off; { Turn off selected waves (usually get errors before
    routine has had a chance to do this). }
end;

procedure com_err_wv(w : integer; s : bufstring);
{ Print error message concerning wave w. }
begin
  com_wr_wv(w, s, COLORHL);
end;

procedure com_ex;
{ Executes commands. }
var
  s : bufstring;
begin
  com.cur := 1; { First word. }
  s := com.ls[com.cur]; { Copy word to convenient variable. }
  if s = '' then { Update screen for blank entry. }
  begin
    UpdateVitals;
    Update(sc.cur);
```

```
  end
  else if s = 'abs' then do_abs
  else if s = 'ac' then do_ac_cc(t0.ac)
  else if s = 'ad' then do_ad
  else if s = 'add' then do_add
  else if s = 'area' then do_wv_info(0, s)
  else if s = 'avg' then do_wv_info(0, s)
  else if s = 'auto' then do_auto
  else if s = 'blank' then do_blank
  else if s = 'bg' then do_bg
  else if s = 'bs' then do_bs
  else if s = 'cal' then do_cal
  else if s = 'cc' then do_ac_cc(t0.cc)
  else if s = 'cd' then do_cd
  else if s = 'col' then _col
  else if s = 'cp' then do_cp
  else if s = 'cpd' then do_cpd
  else if s = 'ctr' then do_wv_info(0, s)
  else if s = 'dis' then do_bs_dis
  else if s = 'disc' then disc
  else if s = 'dly' then _dly
  else if s = 'dos' then do_dos
  else if s = 'dots' then do_dots
  else if s = 'ed' then do_ed
  else if s = 'edgel' then do_wv_info(0, s)
  else if s = 'edger' then do_wv_info(0, s)
  else if s = 'en' then SetEnergyConversion
  else if s = 'ex' then ExitProgram
  else if s = 'exit' then ExitProgram
  else if s = 'fit' then do_fit
  else if s = 'fn' then do_fn
  else if s = 'fu' then FullView(sc.cur)
  else if s = 'fwhm' then do_wv_info(0, s)
  else if s = 'gen' then do_gen
  else if s = 'help' then Help
  else if s = 'ht' then do_wv_info(0, s)
  else if s = 'int' then Integral
  else if s = 'inv' then do_vis(0)
  else if s = 'macro' then do_macro
  else if s = 'mc' then do_mc
  else if s = 'mon' then do_mon
  else if s = 'mv' then do_mv
  else if s = 'nor' then ToggleSaveMode
  else if s = 'nw' then do_nw
  else if s = 'osc' then do_osc
  else if s = 'p' then Pause
  else if s = 'pr' then PrintWave
  else if s = 'q' then ExitProgram
  else if s = 'quit' then ExitProgram
  else if s = 'rd' then do_rd
  else if s = 'rebin' then rebin
  else if s = 'redo' then do_redo
  else if s = 'rescale' then _rescale
  else if s = 'resume' then do_run(0)
  else if s = 'rf' then UpdateAll
  else if s = 'rm' then do_rm
  else if s = 'run' then do_run(1)
  else if s = 'sc' then do_sc
  else if s = 'sech' then do_sech
  else if s = 'sh' then do_sh
  else if s = 'sm' then do_sm
  else if s = 'ss' then do_ss
  else if s = 'stack' then StackWaves
```

```
  else if s = 'stop' then do_stop
  else if s = 'sts' then do_sts
  else if s = 'sv' then do_sv
  else if s = 'sys' then SystemControl
  else if s = 't0' then do_t0
  else if s = 'ts' then do_ts
  else if s = 'tw' then do_tw
  else if s = 'vis' then do_vis(1)
  else if s = 'ws' then do_ws
  else if s = 'wv' then do_wv
  else if s = 'x' then do_x(sc.cur)
  else if s = 'xh' then do_xh
  else if s = 'y' then do_y(sc.cur)
  else if s = 'yoff' then do_yoff
  else if s = 'ysc' then do_ysc
  else if s = 'zm' then Zoom
  else com_err;
end;

procedure com_parse(s : bufstring);
{ Parse string s into words in com.ls[]. }
var
  clast : char;
  i : integer;
begin
  com.cur := 1; { Point to first word. }
  com.sv := 'x'; { Set com.sv <> '' as flag to com_wr routine not to
    bother saving. }

  { First truncate trailing spaces. }
  for i := length(s) downto 1 do
    if s[i] = ' ' then
      s := copy(s, 1, i - 1)
    else
      i := 1; { Get out of loop. }

  { Special case: '!!' = echo last string typed. }
  if s = '!!' then
  begin
    com_wr(com.old, COLORMESS); { Echo last string typed to screen. }
    exit; { Just use last com.ls[] values over again. }
  end;

  com.old := s; { Transfer current string to storage for use by !!. }
  com.num := 1;
  clast := ' ';
  for i := 1 to com_ls_MAX do
    com.ls[i] := '';
  { Perform parse. }
  for i := 1 to length(s) do
    { Advance word counter on first space encountered; others ignored. }
    if s[i] = ' ' then
    begin
      if clast <> ' ' then
      begin
        clast := ' ';
        inc(com.num);
      end;
    end
    else
    begin
      { Check if exceed max words (trick into ending for loop). }
      if com.num > com_ls_MAX then
        begin
          com_wr('Maximum words ' + makestringint(com_ls_MAX) +
            '; ignoring rest of line.', COLORHL);
          exit;
        end
        else
        { Add character to current word. }
        begin
          com.ls[com.num] := com.ls[com.num] + s[i];
          clast := s[i];
        end;
      end;
end;

procedure com_rd;
{ Allows user to type on graphics screen, then parses what was typed into
  com.ls array for further processing by com_exec. Also handles direct
  action keystrokes which can be typed while command is being entered. }
var
  c : char;
  exitflag : boolean; { Flag to exit loop. }
  i : integer; { Char counter. }
  s : bufstring;
begin
  exitflag := false;
  com.ystart := com.tx.cur.y;
  tx_dr(@com.tx); { Print screen. }
  com.sv := ''; { Erase previously saved input. }
  repeat
    Scan; { Take care of active waves. }
    com_wr_sv; { Restore input if it has been saved due to messages
      printed during Scan. }
    tx_wr_ch(@com.tx, '_'); { Write cursor. }
    if keypressed then
    begin
      { Read key. }
      c := readkey;
      if (c >= #32) and (c <= #127) then
      begin
        tx_wr_ch(@com.tx, c);
        if com.tx.cur.x < com.tx.num.x then
          inc(com.tx.cur.x)
        else if com.tx.cur.y < com.tx.num.y then
        begin { Advance to next line. }
          com.tx.cur.x := 1;
          inc(com.tx.cur.y);
        end
        else if com.ystart > 1 then
        begin { Scroll down if didn't begin at top. }
          com.tx.cur.x := 1;
          tx_scr_up(@com.tx);
          dec(com.ystart);
        end; { Otherwise, leave cursor at end of line (dead-ended). }
      end
      else case c of
        BS : begin
          tx_wr_ch(@com.tx, ' ');
          if com.tx.cur.x > 1 then
            dec(com.tx.cur.x)
          else if com.tx.cur.y > com.ystart then
          begin { Move to previous line if we haven't gone past beginning.}
            com.tx.cur.x := com.tx.num.x;
            dec(com.tx.cur.y);
```

```
          end; { Otherwise, cursor doesn't move. }
        end;
        CR : exitflag := true;
        CTRLS: ScanStopAll;
        CTRLZ: YScaleChangeSign;
        ESC : Pause;
        EXTENDED :
          case readkey of
          XARROWLEFT: MoveCursor(0, -1);
          XARROWRIGHT: MoveCursor(0, 1);
          XARROWUP: ChangeCurrentWave(-1);
          XARROWDOWN: ChangeCurrentWave(1);
          XCTRLARROWLEFT: MoveCursor(0, -20);
          XCTRLARROWRIGHT: MoveCursor(0, 20);
          XCTRLEND: YScaleChange(-1);
          XCTRLHOME: YScaleChange(1);
          XCTRLPAGEUP: MoveCursor(1, -20);
          XCTRLPAGEDOWN: MoveCursor(1, 20);
          XDELETE: YScaleChange(-0.1);
          XEND: YScaleChange(-0.01);
          XF2: SystemControl;
          XHOME: YScaleChange(0.01);
          XINSERT: YScaleChange(0.1);
          XPAGEUP: MoveCursor(1, -1);
          XPAGEDOWN: MoveCursor(1, 1);
          end;
        TAB : ToggleCrosshairsWhich;
      end;
      tx_wr_ch(@com.tx, '_');
    end;
  until exitflag;
  { Erase cursor. }
  tx_wr_ch(@com.tx, ' ');

  { Transfer text into s. }
  s := '';
  for i := (com.ystart - 1) * com.tx.num.x + 1 to com.tx.cur.x - 1 +
    (com.tx.cur.y - 1 ) * com.tx.num.x do
    s := s + com.tx.buf^[i];
  { Clean up screen display for next action -- only advance cursor if
    something was typed. }
  if (com.tx.cur.y > com.ystart) or (com.tx.cur.x > 1) then
  begin
    com.tx.cur.x := 1;
    if com.tx.cur.y = com.tx.num.y then
      tx_scr_up(@com.tx)
    else
      inc(com.tx.cur.y);
  end;
  com_parse(s); { Parse text into com.ls[]. }
end;

procedure com_wr(s : string; col : word);
{ Print string s in color col to com.tx screen, first saving user input
  if nothing has been saved. }
var
  i : integer;
begin
  { See if something has been saved. }
  if com.sv = '' then
  begin
    { Save input in com.sv. }
    for i := (com.ystart - 1) * com.tx.num.x + 1 to (com.tx.cur.y - 1) *
      com.tx.num.x + com.tx.cur.x - 1 do
      com.sv := com.sv + com.tx.buf^[i];
    { Erase input lines; move current cursor back to start of input. }
    FillChar(com.tx.buf^[(com.ystart - 1) * com.tx.num.x + 1],
      (com.tx.cur.y - com.ystart + 1) * com.tx.num.x, ' ');
    com.tx.cur.x := 1;
    com.tx.cur.y := com.ystart;
  end;
  { Write message. }
  tx_wr(@com.tx, s, col, 1);
  { Update ystart. }
  com.ystart := com.tx.cur.y;
end;

procedure com_wr_db(s : bufstring);
{ Print debug message s, wait for user to hit key. }
begin
  if debug = 0 then
    exit;
  com_wr(s, COLORDEBUG);
  if readkey = extended then
    readkey;
end;

procedure com_wr_wv(w : integer; s : string; col : word);
{ Print message concerning wave w. }
begin
  com_wr('Wave ' + makestringint(w) + ' ' + s, col);
end;

function com_wr_yn(s : bufstring) : integer;
{ Print message s on com, wait for user response:
  Y (yes): 1
  N (no): 0.
  Also stores result in ynaesc_response variable. }
begin
  com_wr(s + ' (Yes/No)?', COLORHL);
  if macro_override = 1 then
  begin
    com_wr_yn := 1;
    ynaesc_response := 1;
    exit;
  end;
  repeat
    Scan; { Keep active waves happy. }
    case readkey of
    'n', 'N':
      begin
        com_wr_yn := 0;
        ynaesc_response := 0;
        exit;
      end;
    'y', 'Y':
      begin
        com_wr_yn := 1;
        ynaesc_response := 1;
        exit;
      end;
    EXTENDED: readkey; { Handle extended keys. }
    end;
  until false;
end;
```

```
function com_wr_ynaesc(s : bufstring) : integer;
{ If the ynaesc_response variable is 0 or 1, prints message s on com, then
  waits for user to type a key, returning:

  A (all yes): return 2
  Y (yes): return 1
  N (no): return 0
  ESC (abort): return -1.

  However, if ynaesc_response = 2 or -1, will return this value immediately,
  neither printing the message nor waiting for user input. This is how
  the concepts "all" and "abort" are implemented. To clear an "all" or
  "abort" condition, set ynaesc_response = 0 (or 1).

  See also com_wr_yn, which has more limited choices. }
begin
  if (ynaesc_response < 0) or (ynaesc_response > 1) then
  begin
    com_wr_ynaesc := ynaesc_response;
    exit;
  end;
  com_wr(s + ' (Yes/No/All/ESC)?', COLORHL);
  repeat
    Scan; { Keep active waves happy. }
    case readkey of
    ESC:
      begin
        com_wr_ynaesc := -1;
        ynaesc_response := -1;
        exit;
      end;
    'a', 'A':
      begin
        com_wr_ynaesc := 2;
        ynaesc_response := 2;
        exit;
      end;
    'n', 'N':
      begin
        com_wr_ynaesc := 0;
        ynaesc_response := 0;
        exit;
      end;
    'y', 'Y':
      begin
        com_wr_ynaesc := 1;
        ynaesc_response := 1;
        exit;
      end;
    EXTENDED: readkey; { Handle extended keys. }
    end;
  until false;
end;

procedure com_wr_sv;
{ Restore saved input to screen. }
begin
  if com.sv <> '' then
  begin
    tx_wr(@com.tx, com.sv, COLORUSER, 0); { 0: non-scroll mode. }
    com.ystart := com.tx.cur.y - length(com.sv) div com.tx.num.x;
    com.sv := '';
  end;
end;

procedure do_abs;
{ Switch to absolute y mode. }
begin
  if sc.ls[sc.cur].gr.yaxismode = YAXISMODE_RELATIVE then
    ToggleYAxisMode(sc.cur);
end;

procedure do_ac_cc(pos : real);
{ Set up an AC or CC wave centered at position pos. }
begin
  if wv.num = MAXWAVES then
  begin
    com_wr('Memory Full.', COLORHL);
    exit;
  end;
  CreateWave(dt_cor, 0);
  with wv.ls[wv.cur]^ do
  begin
    par.cor.ts.t0 := pos;
    UpdateCORLimits(@par); { Change start & stop. }
  end;
  UpdateVitals;
  Update(sc.cur);
end;

procedure do_ad;
{ Print or change A/D board params. }
var
  ch, chmin, chmax : integer;
  dummy : integer;
  i : integer;
  s : bufstring;
  sel : array[1 .. AD_MAX] of integer; { Selection array. }
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  val(s, chmin, dummy);
  if chmin in [1..AD_MAX] then
  begin
    if com.cur = com.num then
    begin
      com_err;
      exit;
    end;
    chmax := chmin;
    inc(com.cur);
    s := com.ls[com.cur];
  end
  else
  begin
    chmin := 1;
    chmax := AD_MAX;
  end;
  if s = 'gain' then
  begin
    if com.cur = com.num then
```

```
    begin
      for ch := chmin to chmax do
        com_wr('Channel ' + makestringint(ch) + ' gain ' +
          makestringint(ad.ls[ch].gain), COLORMESS);
      exit;
    end;
    inc(com.cur);
    val(com.ls[com.cur], i, dummy);
    if (i = 1) or (i = 2) or (i = 4) or (i = 8) then
      for ch := chmin to chmax do
        ad.ls[ch].gain := i
    else
      com_err;
  end
  else
    com_err;
end;

procedure do_add;
{ Handle addwaves commands. }
var
  i, j, k, dummy : integer;
  w : array[1 .. MAXWAVES] of integer; { Keep track of 1st wavelist. }
begin
  if wv.num = 0 then
    exit;
  if com.num < 3 then
  { Old add functions: turn on/off addwave. }
  begin
    if (addwaves.mode = 1) or (com.cur = com.num) then
      ToggleAddWavesMode(wv.cur)
    else
    begin
      inc(com.cur);
      val(com.ls[com.cur], i, dummy);
      if (i >= 1) and (i <= wv.num) then
        ToggleAddWavesMode(i)
      else
        com_err;
    end;
    exit;
  end;

  { New add function: add (WVLIST) to (WVLIST). }
  wv_sel(1);
  { Copy first wave list to temp array. }
  for i := 1 to wv.num do
    if wv.ls[i]^.sel = 1 then
      w[i] := 1
    else
      w[i] := 0;
  { Ensure that 'to' word is present. }
  if (com.cur = com.num) or (com.ls[com.cur + 1] <> 'to') then
  begin
    com_wr('Format: add (WVLIST) to (WVLIST).', COLORHL);
    exit;
  end;
  inc(com.cur);
  { Get target wave list. }
  wv_sel(1);
  { Remove waves selected in both lists from the target list. }
  for i := 1 to wv.num do
    if (wv.ls[i]^.sel = 1) and (w[i] =1) then
      begin
        wv.ls[i]^.sel := 0;
        com_wr_wv(i, 'removed from target list.', COLORHL);
      end;

  { Do additions. }
  sc_sel_off;
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
      begin
        for j := 1 to par.pt do
          with sc.ls[screen].gr do
          begin
            data[j] := PtoV(i, j, xaxismode_POINTS, yaxismode) *
              sgn(par.yscale); { Need sgn since PtoV uses abs(par.
              yscale). Use POINTS mode to prevent vertical scale
              changing due to nonuniform x axis (e.g. energy space). }
            for k := 1 to wv.num do
              if w[k] = 1 then
                with sc.ls[wv.ls[k]^.screen].gr do
                  data[j] := data[j] + PtoV(k, j, xaxismode_POINTS,
                    yaxismode) * sgn(wv.ls[k]^.par.yscale);
          end;
        { Now reset scaling to default since data has been altered by
          these parameters. }
        par.yscale := 1;
        par.yoffset := 0;
        datasaved := 0;
        parsaved := 0;
        { Disable bs mode so don't get bs dis affecting result. }
        if par.dt = DT_ELE then
          par.ele.bs.mode := 0;
        { Tag screen for update. }
        sc.ls[screen].sel := 1;
      end;
  wv_sel_off;
  UpdateSel;
end;

procedure do_auto;
{ Change auto parameters. }
var
  dummy : integer;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'adv' then
  begin
    if com.cur = com.num then
    begin
      com_wr('Auto.adv ' + makestring(auto.adv / POWFS, VALMAXFS,
        VALDECFS) + ' fs', COLORMESS);
      exit;
    end;
    inc(com.cur);
    s := com.ls[com.cur];
```

```
  if s = '0' then
  begin
    auto.adv := 0;
    exit;
  end;
  val(s, auto.adv, dummy);
  auto.adv := auto.adv * POWFS; { Convert fs to s. }
  roundoff(auto.adv, ts.step / HALFSPEEDOFLIGHT); { Round to
    nearest 1 um. }
  { Check for excessively large step size. }
  if abs(auto.adv) > STAGEMAX / HALFSPEEDOFLIGHT then
    auto.adv := 0;
end
else if s = 'all' then
begin
  if com.cur = com.num then
  begin { Print values of all auto vars. }
    str(auto.bg, s);
    com_wr('Auto.bg ' + s, COLORMESS);
    str(auto.fn, s);
    com_wr('Auto.cor ' + s, COLORMESS);
    str(auto.cor, s);
    com_wr('Auto.fn ' + s, COLORMESS);
    str(auto.gen, s);
    com_wr('Auto.gen ' + s, COLORMESS);
    str(auto.off, s);
    com_wr('Auto.off ' + s, COLORMESS);
    str(auto.rm, s);
    com_wr('Auto.rm ' + s, COLORMESS);
    str(auto.sv, s);
    com_wr('Auto.sv ' + s, COLORMESS);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  { Read value and put in first variable. }
  if s = '0' then
    auto.fn := 0
  else if s = '1' then
    auto.fn := 1
  else
  begin
    com_err;
    exit;
  end;
  { Assign other auto vars. }
  auto.bg := auto.fn;
  auto.cor := auto.fn;
  auto.gen := auto.fn;
  auto.off := auto.fn;
  auto.rm := auto.fn;
  auto.sv := auto.fn;
end
else if s = 'bg' then
  rd_int(auto.bg, 'Auto.bg', 0, 1)
else if s = 'cor' then
  rd_int(auto.cor, 'Auto.cor', 0, 1)
else if s = 'fn' then
  rd_int(auto.fn, 'Auto.fn', 0, 1)
else if s = 'gen' then
  rd_int(auto.gen, 'Auto.gen', 0, 1)
else if s = 'off' then
  rd_int(auto.off, 'Auto.off', 0, 1)
else if s = 'num' then
  rd_int(auto.num, 'Auto.num', 1, 999)
else if s = 'rm' then
  rd_int(auto.rm, 'Auto.rm', 0, 1)
else if s = 'ser' then
begin
  rd_str(auto.ser, 'Auto.ser');
  { Add "-" to end if not already there. }
  if auto.ser[length(auto.ser)] <> '-' then
    auto.ser := auto.ser + '-';
end
else if s = 'sv' then
  rd_int(auto.sv, 'Auto.sv', 0, 1)
else
  com_err;
end;

procedure do_blank;
{ Blank part of waves. }
var
  dummy : integer;
begin
  if com.cur = com.num then
  { Assume just blank current wave, using current limits. }
  begin
    wv_sel(1);
    Blank;
    exit;
  end;
  inc(com.cur);
  if com.ls[com.cur] = 'lim' then
  begin
    if com.cur = com.num then
    { Print current limits. }
    begin
      com_wr('blankmin ' + makestringint(blankmin) + ' blankmax ' +
        makestringint(blankmax), COLORMESS);
      exit;
    end;

    { Read new limits. }
    inc(com.cur);
    if com.cur + 1 > com.num then { Ensure there are two words follow-
      ing 'lim'. }
    begin
      com_err;
      exit;
    end
    else
    begin
      val(com.ls[com.cur], blankmin, dummy);
      val(com.ls[com.cur + 1], blankmax, dummy);
      { Keep in range. }
      if blankmin < 1 then
        blankmin := 1;
      if blankmax > MAXPOINTS then
        blankmax := MAXPOINTS;
    end
  end
  else
    dec(com.cur);
  wv_sel(1);
  Blank;
```

```
end;

procedure do_bg;
{ Print or change bg wave. }
var
  dummy, i : integer;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    do_bg_chg(wv.cur); { Change current wave to bg. }
    exit;
  end;

  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
  { Show current bg wave. }
    com_wr('bs bg ' + makestringint(_bs.bg), COLORMESS)
  else
  begin
    val(s, i, dummy);
    do_bg_chg(i);
  end;
end;

procedure do_bg_chg(i : integer);
{ Change current bg wave to i if allowed. }
begin
  if (i < 1) or (i > wv.num) then
  begin
    com_wr('Wave number out of range.', COLORHL);
    exit;
  end;
  with wv.ls[i]^ do
  begin
    { Forbid change if either:
      1. current wave is bs scanning,
      2. new wave is bs scanning. }
    if ((_bs.bg > 0) and (wv.ls[_bs.bg]^.par.dt = dt_ELE) and
      (wv.ls[_bs.bg]^.par.ele.bs.mode = 1) and (wv.ls[_bs.bg]^.
      scan.mode > 0))
      or
      ((par.dt = dt_ELE) and (par.ele.bs.mode = 1) and (scan.mode >
      0))
      then
    begin
      com_wr_wv(i, 'active! No change.', COLORHL);
      exit;
    end;
    if par.dt <> dt_ELE then
    begin
      com_wr_wv(i, 'not ELE! No change.', COLORHL);
      exit;
    end;
    _bs.bg := i;
    par.ele.bs.mode := 1; { Turn on mode if off. }
  end;
  DrawWaveData; { Update 'B' flag. }
end;

procedure do_bs;
{ Handle bs commands. }
var
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    do_bs_mode(1);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '0' then
    do_bs_mode(0)
  else if s = '?' then
    do_bs_mode(-1)
  else if s = 'adapt' then
    BsAdapt
  else if s = 'dis' then
    do_bs_dis
  else if s = 'off' then
    do_bs_mode(0)
  else if s = 'on' then
    do_bs_mode(1)
  else if is_sel(s, 1) then
  begin
    dec(com.cur); { Back up before 1st sel word. }
    do_bs_mode(1);
  end
  else
    com_err;
end;

procedure do_bs_dis;
  { Print or change dis mode. }
var
  i : integer;
  s : bufstring;
begin
  if com.cur = com.num then
    _bs.dis := 1
  else
  begin
    inc(com.cur);
    s := com.ls[com.cur];
    if s = '?' then
    begin
      com_wr('bs dis ' + makestringint(_bs.dis), COLORMESS);
      exit;
    end
    else if (s = '0') or (s = 'off') then
      _bs.dis := 0
    else if (s = '1') or (s = 'on') then
      _bs.dis := 1
    else
    begin
      com_err;
      exit;
    end;
  end;
  sc_sel_off;
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if (par.dt = dt_ELE) and (par.ele.bs.mode = 1) then
```

```
        sc.ls[screen].sel := 1;
  UpdateSel;
end;

procedure do_bs_mode(i : integer);
{ Print (i = -1) or change (i = 0, 1) par.ele.bs.mode of waves. }
var
  j : integer;
begin
  wv_sel(1); { Get wave list. }
  if com.cur <> com.num then
  { Should not be additional words at end. }
  begin
    com_err;
    exit;
  end;
  for j := 1 to wv.num do
    with wv.ls[j]^ do
      if sel = 1 then
      begin
        if i = -1 then
          com_wr_wv(j, 'bs mode ' + makestringint(par.ele.bs.mode),
            COLORMESS)
        else if par.dt <> DT_ELE then
          com_wr_wv(j, 'not ELE! No change.', COLORHL)
        else if scan.mode > 0 then
          com_wr_wv(j, 'scanning! No change.', COLORHL)
        else
        begin
          par.ele.bs.mode := i;
          { Change bg wave if currently invalid. }
          with wv.ls[_bs.bg]^ do
            if (_bs.bg < 1) or (_bs.bg > wv.num) or (par.dt <>
              dt_ELE) or (par.ele.bs.mode = 0) then
              _bs.bg := j;
        end;
      end;
  wv_sel_off;
  DrawWaveData; { Erase selection tags, update bs tags. }
end;

procedure do_cal;
begin
  if com.cur < com.num then
  begin
    inc(com.cur);
    case wv.ls[wv.cur]^.par.dt of
    dt_ELE: case com.ls[com.cur][1] of
     '1': CalibEnergy(1);
     '2': CalibEnergy(2);
     else com_err;
     end;
    dt_MAS: case com.ls[com.cur][1] of
      '1': CalibMass1;
      '2': CalibMass2;
      else com_err;
      end;
    else com_err;
    end;
  end
  else
    com_err;
end;
```

```
procedure do_cd;
{ Print or change current directory. }
var
  i,j,k : integer; { counters }
  s : bufstring; { user input buffer }
  tdir, tdir2 : bufstring; { Temporary directories. }
begin
  if com.cur = com.num then
  begin
    com_wr(dir, COLORMESS);
    exit;
  end;
  tdir := dir; { Copy directory for manipulation. }
  inc(com.cur);
  s := com.ls[com.cur];
  if s <> '' then { if user types nothing, no change }
  begin
    if s[1] = '.' then
     { '.' indicates user wants extension of current directory }
      for i := 2 to length(s) do
        tdir := tdir + s[i]
    else
      begin
        { remove subdirectories }
        if (s[1] = '-') then
          begin
            j := 1;
            tdir2 := tdir; { restore if necessary }
            repeat
              i := length(tdir) - 1;
              repeat
                i := i - 1;
              until ((tdir[i] = '\') or (i=1));
              delete(tdir, i+1, length(tdir)-i);
              j:=j+1;
            until ((s[j]<>'-') or (i=1));
            for k := j to length(s) do
              tdir := tdir + s[k];
            if (i=1) then
              tdir := tdir2;
          end
        else
        { manual input of new directory }
          begin
            tdir := ' ';
            for i := 2 to length(s) do
              tdir := tdir + ' ';
            for i := 1 to length(s) do
              tdir[i] := s[i];
          end;
      end;
  end;
  UpdateFilenames(tdir); { Assigns tdir to dir inside procedure. }
  DrawWaveData; { fn representation may change. }
end;

procedure do_cp;
{ Copy waves. }
var
  tempnumwaves : integer;
  w : integer;
begin
```

```
  wv_sel(1);
  if wv.sel = 0 then
    exit;
  tempnumwaves := wv.num;
  for w := 1 to tempnumwaves do
    with wv.ls[w]^ do
      if sel = 1 then
      begin
        if wv.num = MAXWAVES then
        begin
          com_wr('Memory full.', COLORHL);
          w := tempnumwaves; { Trick to get out of loop. }
        end;
        inc(wv.num);
        wv.ls[wv.num]^ := wv.ls[w]^;
        InitializeWave(wv.num); { Set certain parameters to defaults. }
        { Now change a couple back: }
        wv.ls[wv.num]^.datasaved := wv.ls[w]^.datasaved;
        wv.ls[wv.num]^.lines := wv.ls[w]^.lines;
        wv.ls[wv.num]^.parsaved := wv.ls[w]^.parsaved;
        wv.ls[wv.num]^.par.sh := wv.ls[w]^.par.sh;
      end;
  wv_sel_off; { Turn off wave selections. }
  UpdateVitals;
  Update(sc.cur);
end;

procedure do_cpd;
{ Print or change current print directory. }
var
  i,j,k : integer;                    { counters }
  s : bufstring;                      { user input buffer }
  tempdir : bufstring;                { for restoring directory }
begin
  if com.cur = com.num then
  begin
    com_wr(printdir, COLORMESS);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s <> '' then { if user types nothing, no change }
  begin
    if s[1] = '.' then
                { '.' indicates user wants extension of current directory }
      for i := 2 to length(s) do
        printdir := printdir + s[i]
    else
      begin
        { remove subdirectories }
        if (s[1] = '-') then
          begin
            j := 1;
            tempdir := printdir; { restore if necessary }
            repeat
              i := length(printdir) - 1;
              repeat
                i := i - 1;
              until ((printdir[i] = '\') or (i=1));
              delete(printdir, i+1, length(printdir)-i);
              j:=j+1;
            until ((s[j]<>'-') or (i=1));
            for k := j to length(s) do
              printdir := printdir + s[k];
            if (i=1) then
              printdir := tempdir;
          end
        else
        { manual input of new directory }
          begin
            printdir := ' ';
            for i := 2 to length(s) do
              printdir := printdir + ' ';
            for i := 1 to length(s) do
              printdir[i] := s[i];
          end;
      end;
  end;
end;

procedure do_dos;
{ Interfact to ShellToDOS code. }
begin
  TextMode;
  ShellToDOS;
  DrawAll;
end;

procedure do_dots;
{ Change dot size. }
var
  dummy : integer;
  i : integer;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    str(dotradius, s);
    com_wr('Dot radius = ' + s, COLORMESS);
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], i, dummy);
  if (i < 0) or (i > MAXDOTSIZE) then
  begin
    com_err;
    exit;
  end;
  dotradius := i;
  for i := 1 to sc_MAX do
    sc.ls[i].sel := 0;
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if (on = 1) and (lines = 0) then
        sc.ls[screen].sel := 1;
  for i := 1 to sc_MAX do
    if sc.ls[i].sel = 1 then
      DrawScreen(i);
  DrawVitals;
end;

procedure do_ed;
{ Edit current or specified wave, or defaults ("df"). }
var
  dummy : integer;
  i : integer;
```

```
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    if wv.num > 0 then
      ChangePar(wv.cur);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'df' then
  begin
    ChangePar(0);
    exit;
  end
  else
  begin
    val(s, i, dummy);
    if (i >= 1) and (i <= wv.num) then
    begin
      ChangePar(i);
      exit;
    end;
  end;
  com_err;
end;

procedure do_fit;
var
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    FitY(dt_MIN - 1);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'area' then
    AreaFitY
  else if s = 'bs' then
    FitBs
  else
    com_err;
end;

procedure do_fn;
{ Renames wave names. Two modes:
  EXACT MODE (fn x): Renames waves with same filename, does not need to
    adhere to standard filename convention.
  NORMAL MODE (fn): Renames waves starting with number NUM. If more
    than one wave specified, increments number for subsequent waves.
  STEP MODE (fn num step num): }

var
  aser : bufstring;
  anum : integer;
  dash : boolean;
  dummy : integer;
  ext : bufstring;
  i : integer;
  s : bufstring;
  step : integer;

begin
  if (wv.num = 0) or (com.cur = com.num) then
  begin
    com_err;
    exit;
  end;
  { Check for 'x' in second word. }
  inc(com.cur);
  if com.ls[com.cur] = 'x' then
  begin
    do_fn_x;
    exit;
  end;

  { Break filename into series, number and extension parts. }
  s := com.ls[com.cur];
  intelligent_filename(s, aser, anum, ext);

  { Check that num is >= 0. }
  if anum < 0 then
  begin
    com_err;
    exit;
  end;

  if com.cur < com.num then
  begin
    inc(com.cur);
    if com.ls[com.cur] = 'step' then
    begin
      if com.cur = com.num then
      begin
        com_err;
        exit;
      end;
      inc(com.cur);
      val(com.ls[com.cur], step, dummy);
      if step <= 0 then
      begin
        com_err;
        exit;
      end;
    end
    else
    begin
      dec(com.cur);
      step := 1;
    end;
  end;

  { Now we go thru waves, renaming their filenames if selected. We keep
    the same file extension they had before. }
  auto.ser := aser;
  auto.num := anum;
  wv_sel(1);
  if wv.sel = 0 then
    exit;
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
      begin
        str(auto.num, s);
        if ext = '' then
```

```
          par.fn := auto.ser + s + '.' + get_extension(par.fn) { Keep
            old extension. }
        else
          par.fn := auto.ser + s + '.' + ext; { Use new extension. }
        inc(auto.num,step);
      end;
  UpdateFilenames(dir); { Update filenames if path changed. }
  wv_sel_off;
  DrawWaveData;
end;

procedure do_fn_x;
{ Renames waves with exact filename (no conventions heeded). }
var
  i : integer;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur]; { Record filename. }
  wv_sel(1);
  if wv.sel = 0 then
    exit;
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
        par.fn := s;
  UpdateFilenames(dir); { Update filenames if path changed. }
  wv_sel_off;
  DrawWaveData;
end;

procedure do_fpes;
{ Set up multiple waves for fpes experiment. Format of file is:
  - bg wave (0 to turn off bs mode for all waves).
  - Delay times (fs) for each wave desired. Will use default ele
    parameters (t0, number of shots, etc.) for everything else. }
var
  bg_temp : integer;
  delay : real;
  f : text; { File variable. }
  i : integer;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if get_extension(s) = '' then
    s := s + '.' + FPES_EXT;
{  if not FullPath(s) then
    s := dir + s; }
  if not FileExists(s) then
  begin
    com_wr(s + ' does not exist!', COLORHL);
    exit;
  end;

  if wv.num > 0 then
  begin
    if com_wr_yn('Load FPES template. Erase waves') = 1 then
    begin
      for i := 1 to wv.num do
        wv.ls[i]^.sel := 1;
      EraseWaves;
    end;
  end;
  assign(f, s);
  reset(f);
  readln(f, bg_temp);
  if bg_temp > 0 then { Shift up location of bg wave if other waves
    present. }
    bg_temp := bg_temp + wv.num;
  if bg_temp > MAXWAVES then
  begin
    com_wr('Bg wave number exceeds maximum. Aborting.', COLORHL);
    exit;
  end;
  _bs.bg := bg_temp;
  while (wv.num < MAXWAVES) and (not eof(f)) do
  begin
    readln(f, delay);
    delay := delay * POWFS;
    CreateWave(dt_ELE, 0); { Use default values. }
    with wv.ls[wv.cur]^.par do
    begin
      if _bs.bg = 0 then
        ele.bs.mode := 0
      else
        ele.bs.mode := 1;
      ele.dly := delay;
      { Fix other parameters to ensure all is ok. }
      ele.ts.pos := ele.dly * HALFSPEEDOFLIGHT + ele.ts.t0;
      roundoff(ele.ts.pos, ts.step);
      limit(ele.ts.pos, STAGEMIN, STAGEMAX);
      { Update dly again. }
      ele.dly := (ele.ts.pos - ele.ts.t0) / HALFSPEEDOFLIGHT;
    end;
  end;
  close(f);
  UpdateAll;
end;

procedure do_gen;
{ Generate new wave, of optionally specified datatype, using auto
  filename. }
var
  i : integer;
  s : bufstring;
begin
  if com.cur = com.num then
    CreateWave(0, 0)
  else
  begin
    inc(com.cur);
    s := com.ls[com.cur];
    for i := dt_MIN to dt_MAX do
      if s = dt_NAME[i] then
      begin
```

```
        CreateWave(i, 0);
        i := dt_MAX;
      end;
  end;
  UpdateVitals;
  Update(sc.cur);
end;

procedure do_macro;
{ Read macro file from disk and execute commands. }
var
  f : text;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
{  if not FullPath(s) then
    s := dir + s; }
  if not FileExists(s) then
  begin
    com_wr(ERR_FILENOTFOUND, COLORHL);
    exit;
  end;
  assign(f, s);
  reset(f);
  macro_override := 1;
  while not eof(f) do
  begin
    { See if user wants to abort. }
    if keypressed then
      case readkey of
      CTRLC:
        begin
          close(f);
          macro_override := 0;
          exit;
        end;
      EXTENDED: readkey;
      end;
    readln(f, s); { Read command line. }
    com_wr(s, COLORMACRO); { Echo to screen. }
    com_parse(s); { Parse into com.ls[]. }
    com_ex; { Execute command. }
  end;
  close(f);
  macro_override := 0;
end;

procedure do_mc;
{ Handle mass calculator (mc) commands. }
var
  i, j, lim : integer;
  r : real;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    UpdateMC;
    com_wr(mc.s, COLORMESS);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'auto' then
  begin
    if com.cur = com.num then
    begin { Print current mode state. }
      str(mc.auto, s);
      com_wr('mc auto ' + s, COLORMESS);
      exit;
    end;
    inc(com.cur);
    s := com.ls[com.cur];
    if s = '0' then mc.auto := 0
    else if s = '1' then mc.auto := 1
  end
  else if s = 'ls' then { Print list to screen. }
  begin
    if mc.fn = '' then
    begin
      com_wr('No mc file loaded.', COLORHL);
      exit;
    end;
    Textmode;
    writeln('Filename ', mc.fn, ' Number of elements ', mc.num);
    for j := 0 to mc.num div MC_LINESPERSCREEN do
    begin
      if j < mc.num div MC_LINESPERSCREEN then
        lim := MC_LINESPERSCREEN
      else
        lim := mc.num mod MC_LINESPERSCREEN;
      for i := 1 to lim do
        with mc.ls[i + j * MC_LINESPERSCREEN] do
          writeln('Mass ', makestring(m, VALMAX, VALDEC), ' Min ',
            min, ' Max ', max, '  Label ', s);
      write('Press any key to continue.');
      if readkey = EXTENDED then
        readkey;
      writeln;
    end;
    DrawAll;
  end
  else if s = 'rd' then
  begin
    mc.fn := 'x';
    mc_rd(com.ls[com.cur + 1]);
  end
  else if s = 'sens' then
  begin
    if com.cur = com.num then
    begin { Print sensitivity. }
      com_wr('mc sens ' + makestring(mc.sens, VALMAX, VALDEC),
        COLORMESS);
      exit;
    end;
    inc(com.cur);
    val(com.ls[com.cur], r, i);
    if r <= 0 then
      com_err
    else
      mc.sens := r;
```

```
  end
  else com_err;
end;

procedure do_mon;
{ Handle commands for mon(itor) command. }
var
  dummy : integer;
  i : integer;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    com_wr('mon.w ' + makestringint(mon.w), COLORMESS);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '0' then
  begin
    mon.w := 0;
    DrawWaveData; { Update wave indicator. }
  end
  else if s = 'bins' then
  begin
    if com.cur = com.num then
    begin
      com_wr('mon.bins ' + makestringint(mon.bins), COLORMESS);
      exit;
    end;
    inc(com.cur);
    s := com.ls[com.cur];
    val(s, mon.bins, dummy);
    if mon.bins < 1 then
      mon.bins := 1
    else if mon.bins > MAXPOINTS then
      mon.bins := MAXPOINTS;
  end
  else if s = 'tot' then
    com_wr('mon.tot ' + makestringint(mon.tot), COLORMESS)
  else
  begin
    val(s, i, dummy);
    if (i < 1) or (i > wv.num) then
    begin
      com_wr('Wave number out of range.', COLORHL);
      exit;
    end;
    if (wv.ls[i]^.scan.mode > 0) or (addwaves.w = i) then
    begin
      com_wr_wv(i, 'active! No change.', COLORHL);
      exit;
    end;
    if (wv.ls[i]^.par.dt <> dt_ELE) then
    begin
      com_wr_wv(i, 'not ELE datatype! No change.', COLORHL);
      exit;
    end;
    mon.w := i;
    DrawWaveData; { Update wave indicator. }
  end;
end;
```

```
procedure do_mv;
{ Change to mv scale for all visible mas waves. }
var
  i : integer;
  ytemp : real;
begin
  sc_sel_off;
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if (on = 1) and (par.dt = dt_MAS) then
      begin
        ytemp := par.yscale; { Save before changing. }
        par.yscale := par.mas.vert / TEK_YPTSPERDIV;
        if (yoffsetrescale = 1) and (ytemp <> 0) then
          par.yoffset := par.yoffset * par.yscale / ytemp;
        sc.ls[screen].sel := 1;
      end;
  UpdateVitals;
  for i := 1 to sc.num do
    with sc.ls[i] do
      if sel = 1 then
      begin
        gr.yaxismode := YAXISMODE_RELATIVE;
        Update(i);
      end;
end;

procedure do_nw;
{ Generate new wave, of optionally specified datatype, using filename of
  specified wave. }
var
  d, dummy, i, n : integer;
  s : bufstring;
begin
  { Find number of waves to generate. }
  inc(com.cur);
  s := com.ls[com.cur];
  val(s, n, dummy);
  if n > 0 then
    inc(com.cur) { Go to next word. }
  else
    n := 1;
  { Find datatype (or leave as d = 0). }
  begin
    s := com.ls[com.cur];
    d := 0;
    for i := dt_MIN to dt_MAX do
      if s = dt_NAME[i] then
      begin
        d := i;
        i := dt_MAX; { Get out of loop. }
      end;
  end;
  if d = 0 then
    dec(com.cur); { Back up to previous word since nothing was recog-
      nized. }
  for i := 1 to n do
    if wv.num = MAXWAVES then
      i := n
    else if wv.num = 0 then
      CreateWave(d, 0) { No previous wave to base on: use defaults. }
    else
      CreateWave(d, wv.cur);
```

```
  UpdateVitals;
  Update(sc.cur);
end;

procedure do_osc;
{ Handle virtual oscilloscope functions. }
const
  OSC_ON = 'Osc on.';
  OSC_OFF = 'Osc off.';
var
  dummy : integer;
  i : integer;
  s : bufstring;
begin
  with osc do
  begin
    if com.cur = com.num then
    begin
      mode := 1 - mode;
      if mode = 0 then
      begin
        ADoff(ch);
        com_wr(OSC_OFF, COLORMESS);
      end
      else
      begin
        AD.ls[ch].on := 1;
        com_wr(OSC_ON, COLORMESS);
      end;
      exit;
    end;
    inc(com.cur);
    s := com.ls[com.cur];
    if s = 'ch' then
    begin
      if com.cur = com.num then
      begin
        com_wr('Osc channel = ' + makestringint(ch), COLORMESS);
        exit;
      end;
      inc(com.cur);
      val(com.ls[com.cur], i, dummy);
      if (i >= 1) and (i <= AD_MAX) then
      begin
        if mode = 0 then { If osc was off, print message to alert that
          now on. }
          com_wr(OSC_ON, COLORMESS);
        mode := 0; { Must turn off so channel can be freed. }
        ADoff(ch);
        ch := i;
        Ad.ls[ch].on := 1;
        mode := 1; { Turn on again. }
      end
      else
        com_err;
    end
    else if s = 'fu' then
      with sc.ls[scr].gr do
      begin
        v1lim := 2048;
        v2lim := 4095;
        yfullmode := 0;
        UpdateVitals;
        Update(scr);
      end
    else if s = 'sc' then
    begin
      if com.cur = com.num then
      begin
        com_wr('Osc screen = ' + makestringint(scr), COLORMESS);
        exit;
      end;
      inc(com.cur);
      val(com.ls[com.cur], i, dummy);
      if (i >= 1) and (i <= sc.num) then
      begin
        scr := i;
        { Turn on if not on. }
        if mode = 0 then { If osc was off, print message to alert that
          now on. }
          com_wr(OSC_ON, COLORMESS);
        Ad.ls[ch].on := 1;
        mode := 1;
      end
      else
        com_err;
    end
    else
      com_err;
  end;
end;

procedure do_rd;
{ Read waves. Checks for existence. Terminates with error if there are
  more files than available waves in memory. }
var
  abort : boolean; { Flag to abort procedure. }
  anum : integer;
  aser : bufstring;
  dummy : integer;
  ext : bufstring;
  s : bufstring;

  procedure do_rd_exit; forward;
  procedure do_rd_fn; forward;
  procedure do_rd_wv; forward;
  procedure do_rd_x; forward;

  procedure do_rd_exit;
  { Tidy up before exiting. }
  begin
    UpdateVitals;
    Update(sc.cur); { Since can only load waves into current screen,
      this works. }
  end;

  procedure do_rd_fn;
  { Assemble and check for existence of filename. }
  var
    i : integer;
  begin
    str(auto.num, s);
    if ext = '' then
    begin
      s := auto.ser + s;
      if not FullPath(s) then
```

```
      s := dir + s;
    { Try all dt extensions: }
    i := dt_MIN;
    while (i <= dt_MAX) and (not FileExists(s + '.' + dt_NAME[i])) do
      inc(i);
    if i <= dt_MAX then
      s := s + '.' + dt_NAME[i];
  end
  else
  begin
    s := auto.ser + s + '.' + ext; { Use new extension. }
    if not FullPath(s) then
      s := dir + s;
  end;
  do_rd_wv;
  inc(auto.num); { Prepare for possible increments. }
end;

procedure do_rd_wv;
{ Call readwave procedure. }
var
  w : integer;
begin
  if keypressed then
    case readkey of
    ESC : { Get out of read. }
      begin
        com_wr('Aborted.', COLORMESS);
        abort := true;
        exit;
      end;
    EXTENDED : readkey; { Clean out buffer in case of extended key. }
    end;
  if not FileExists(s) then
  begin
    com_wr(s + ' does not exist', COLORHL);
    exit;
  end;
  if wv.num = MAXWAVES then
  begin
    com_wr('Memory full.', COLORHL);
    abort := true;
    exit;
  end;
  begin
    w := wv.num + 1;
    if ReadWave(s, w, TRUE) = FALSE then
      com_wr('Bad load on ' + s, COLORHL)
    else
      InitializeWave(w); { Change other parameters, update wv.num,
        etc. }
  end;
end;

procedure do_rd_x;
{ Literal read. }
begin
  while (com.cur < com.num) and (abort = false) do
  begin
    inc(com.cur);
    s := com.ls[com.cur];
    if not FullPath(s) then
      s := dir + s;
    do_rd_wv;
  end;
  do_rd_exit;
end;

BEGIN
  abort := false;
  ext := ''; { Prevent disaster by presetting this. }
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'fpes' then
  begin
    do_fpes;
    exit;
  end
  else if s = 'ws' then
  begin
    if com.cur = com.num then
      ws_rd(WS_FN_DF)
    else
      ws_rd(com.ls[com.cur + 1]);
    UpdateAll;
    exit;
  end
  else if s = 'x' then { Literal filename mode. }
  begin
    do_rd_x;
    exit;
  end;
  com_wr('Read waves. Press ESC to abort.', COLORMESS);
  dec(com.cur);
  while (com.cur < com.num) and (abort = false) do
  begin
    inc(com.cur);
    s := com.ls[com.cur];
    if (s = 'to') then { Flag that next word is a file number for
      us to read to; assumes auto.ser, auto.num already set. }
    begin
      if (com.cur = com.num) then { Must be a number following 'to'. }
      begin
        com_err;
        do_rd_exit;
        exit;
      end;
      inc(com.cur);
      s := com.ls[com.cur];
      val(s, anum, dummy); { Get number. }
      while (auto.num <= anum) and (abort = false) do
        do_rd_fn;
    end
    else
    begin
      intelligent_filename(s, aser, anum, ext); { Analyze filename. }
      { Check that num is >= 0. }
      if anum < 0 then
      begin
        com_err;
        do_rd_exit;
```

```
        exit;
      end;
      auto.ser := aser;
      auto.num := anum;
      do_rd_fn;
    end;
  end;
  do_rd_exit;
end;

procedure do_redo;
{ Erase most recent scan from a wave, decreases scan counter and re-
  starts scan if off. Note waves may be currently scanning.}
var
  i, j : integer;
begin
  wv_sel(1);
  if com.cur < com.num then
  begin
    inc(com.cur); { Put cursor on mystery word. }
    com_err;
    exit;
  end;

  sc_sel_off;
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if (sel = 1) and (par.scan > 0) then
      begin
        sc.ls[screen].sel := 1; { Flag screen for update. }
        for j := 1 to par.pt do
        begin
          data[j] := data[j] - tmp[j];
          tmp[j] := 0; { Erase tmp to prevent disastrous multiple
            redos. }
        end;
        { Special handling for bs electron waves. }
        if (par.dt = dt_ELE) and (par.ele.bs.mode = 1) then
        begin
          par.ele.bs.tot := par.ele.bs.tot - par.ele.bs.last;
          par.ele.bs.last := 0;
        end;
        dec(par.scan);
        datasaved := 0;
        parsaved := 0;
      end;
  wv_sel_off;
  UpdateSel;
end;

procedure do_rm;
{ Remove waves. }
begin
  { Get list of waves to erase. }
  wv_sel(1);
  { Exit if extra words after wave list. }
  if com.cur < com.num then
  begin
    inc(com.cur); { Put cursor on mystery word. }
    com_err;
    exit;
  end;
  if wv.sel = 1 then
    { Get permission before erasing. }
    if AreYouSure then
    begin
      EraseWaves;
      UpdateSel;
    end;
end;

procedure do_run(blank : integer);
{ Activates wave(s) for scanning. blank is a flag to erase existing
  data and reset scan number (blank just passed on to ScanInit, which
  does the actual initializing). }

  procedure check(w : integer);
  { Internal function: check that wave satisfies criteria for scanning.
    }
  var
    s : bufstring;
  begin
    str(w, s);
    s := 'Wave ' + s + ' ';
    with wv.ls[w]^ do
    begin
      { Check if already scanning. }
      if scan.mode > 0 then
      begin
        com_wr(s + 'already scanning.', COLORHL);
        exit;
      end;
      { Check if in addwaves mode. }
      if (addwaves.mode = 1) and (addwaves.w = wv.cur) then
      begin
        com_wr(s + 'in add waves mode.', COLORHL);
        exit;
      end;
      { Check that number of points is compatible with MCS in ele
        scans. }
      if (par.dt = DT_ELE) and (par.pt <> MCS_PT[MCS_PT_MIN]) then
      begin
        com_wr(s + 'pt not compatible.', COLORHL);
        exit;
      end;
      { If there is already data in wave make sure user wants to over-
        write it. }
      if datasaved = 0 then
      begin
        if com_wr_ynaesc(s + 'not saved. Overwrite') < 1 then
        begin
          com_wr(s + 'scan aborted.', COLORMESS);
          exit;
        end;
        com_wr(s + 'overwriting.', COLORMESS);
      end;
      ScanInit(w, blank); { Initialize scan. }
      sc.ls[screen].sel := 1; { Tag screen for update afterward. }
    end;
  end;

var
  f : boolean;
  i : integer;
begin
  if wv.num = 0 then
```

```
    exit;
  wv_sel(1);
  if com.cur < com.num then { Makes no sense if more words at end. }
  begin
    inc(com.cur); { Point to trouble word. }
    com_err;
    exit;
  end;
  sc_sel_off;
  ynaesc_response := 0; { Initialize response var. }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
        check(i);
  wv_sel_off;

  { Check bg wave status if bs waves present. }
  if bs_on then
  begin
    { Check range on background wave. }
    if (_bs.bg < 1) or (_bs.bg > wv.num) then
    begin
      com_wr('Background wave out of range.', COLORHL);
      exit;
    end;
    { Check datatype of bg wave. }
    with wv.ls[_bs.bg]^ do
      if (par.dt <> dt_ELE) or (par.ele.bs.mode = 0) then
      begin
        com_wr_wv(_bs.bg, 'bg not compatible.', COLORHL);
        exit;
      end;
    { Figure out if any waves scanning; set scanwave to 0 if not. }
    f := false;
    for i := 1 to wv.num do
      if wv.ls[i]^.scan.mode > 0 then
        f := true;
    if f = false then
      scanwave := 0;

    { If not in middle of a bs scan, reset bs engine to beginning
      (_bs.fg = 0). }
    with wv.ls[scanwave]^ do
      if (scanwave = 0) or (par.scan = 0) or ((par.dt = dt_ELE) and
        (par.ele.bs.mode = 0)) then
        _bs.fg := 0;
  end;

  mcs.new := 1; { Flag to send all MCS commands the first time. }

  { Wait for scan routine to update screen. }
  { Update affected screens. }
{  UpdateVitals;
  for i := 1 to sc.num do
    if sc.ls[i].sel = 1 then
      Update(i);
  sc_sel_off; }
end;

procedure do_sc;
{ Screen commands. }
var
  bdy : bdy_type; { Temp. screen limits variable. }
  dummy : integer; { For val. }
  i : integer; { Holding val results. }
  s, s2 : bufstring; { Current word. }
  temp : integer;
begin
  if com.cur = com.num then
  begin { Print current screen. }
    str(sc.cur, s);
    com_wr('Current screen = ' + s, COLORMESS);
    exit;
  end;
  sc_sel;
  if sc.sel = 0 then
    exit; { Nothing to do. }
  if com.cur = com.num then { Assume just switch screens. Take first
    screen number from list. }
  begin
    for i := 1 to sc.num do
      if sc.ls[i].sel = 1 then
      begin
        EraseCursor(sc.cur); { Erase old cursor first. }
        temp := sc.cur;
        sc.cur := i;
        str(temp, s);
        str(sc.cur, s);
        DrawTitle(temp);  { Change colors of old and new screen titles. }
        DrawTitle(sc.cur);
        DrawCursor(sc.cur); { Draw new cursor. }
        exit;
      end;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'bdy' then
  { Report overall screen boundaries and exit. }
  begin
    str(sc.bdy[1].x, s2);
    s := 'sc.bdy x ' + s2;
    str(sc.bdy[2].x, s2);
    s := s + ' ' + s2;
    str(sc.bdy[1].y, s2);
    s := s + ' y ' + s2;
    str(sc.bdy[2].y, s2);
    s := s + ' ' + s2;
    com_wr(s, COLORMESS);
    exit;
  end
  else if s = 'fu' then { Full mode. }
  begin
    for i := 1 to sc.num do
      if sc.ls[i].sel = 1 then
        FullView(i);
  end
  else if s = 'num' then { Report total number of screens. }
  begin
    str(sc.num, s);
    com_wr('Number of screens = ' + s, COLORMESS);
    exit;
  end
  else if s = 'rf' then { Refresh. }
  begin
    UpdateVitals;
    for i := 1 to sc.num do
```

```
    if sc.ls[i].sel = 1 then
      Update(i);
end
else if s = 'rm' then { Remove screens. }
begin
  if com_wr_yn('Are you sure') = 0 then
    exit;
  for i := sc.num downto 1 do
    with sc.ls[i] do
      if (sel = 1) and (sc.num > 1) then { Cannot kill only screen. }
      begin
        if i < sc.num then
          for temp := i + 1 to sc.num do { Move down other screens. }
            sc.ls[temp - 1] := sc.ls[temp];
        dec(sc.num);
        if sc.cur >= i then { Move pointer to correct screen. }
          dec(sc.cur);
        if sc.cur = 0 then { Keep from disappearing. }
          sc.cur := 1;
        for temp := 1 to wv.num do { Move waves to correct screen.}
          with wv.ls[temp]^ do
          begin
            if screen >= i then
              dec(screen);
            if screen = 0 then
              screen := 1;
          end;
        if osc.scr >= i then { Move osc to correct screen. }
          dec(osc.scr);
        if osc.scr = 0 then
          osc.scr := 1;
      end;
  UpdateAll; { Refresh screen. }
  exit;
end
else if s = 'ss' then
begin
  if sc.sel = 1 then
    for i := 1 to sc.num do
      if sc.ls[i].sel = 1 then
      begin
        do_sc_ss(i);
        exit;
      end;
  com_err;
  exit;
end
else if s = 'sz' then
begin
  if com.cur = com.num then
  { Report current screen sizes. }
  begin
    for i := 1 to sc.num do
      with sc.ls[i] do
        if sel = 1 then
        begin
          str(i, s2);
          s := 'Screen ' + s2;
          str(gr.bdy[1].x, s2);
          s := s + ' x ' + s2;
          str(gr.bdy[2].x, s2);
          s := s + ' ' + s2;
          str(gr.bdy[1].y, s2);
          s := s + ' y ' + s2;
          str(gr.bdy[2].y, s2);
          s := s + ' ' + s2;
          com_wr(s, COLORMESS);
        end;
    exit;
  end;
  { Check there are exactly four more entries. }
  if com.cur + 4 <> com.num then
  begin
    com_err;
    exit;
  end;
  { Read in values. }
  val(com.ls[com.cur + 1], bdy[1].x, dummy);
  val(com.ls[com.cur + 2], bdy[2].x, dummy);
  val(com.ls[com.cur + 3], bdy[1].y, dummy);
  val(com.ls[com.cur + 4], bdy[2].y, dummy);
  { Check that values are all valid. }
  if (bdy[1].x < sc.bdy[1].x) or (bdy[2].x > sc.bdy[2].x) or (bdy[1].y
    < sc.bdy[1].y) or (bdy[2].y > sc.bdy[2].y) or (bdy[1].x >=
    bdy[2].x) or (bdy[1].y >= bdy[2].y) then
  begin
    com_wr('Bad screen limits.', COLORHL);
    exit;
  end;
  { Resize screens and update everything. }
  for i := 1 to sc.num do
    with sc.ls[i] do
      if sel = 1 then
      begin
        gr.bdy := bdy; { Copy temp. bdy to real thing. }
        sc_resize(i);
      end;
  UpdateAll;
end
else if s = 'ti' then { Title. }
begin
  if com.cur = com.num then
  { Print titles. }
  begin
    for i := 1 to sc.num do
      with sc.ls[i] do
        if sel = 1 then
        begin
          str(i, s);
          s := 'Screen ' + s + ' ';
          if ti.on = 1 then
            com_wr(s + 'title = ' + ti.s, COLORMESS)
          else
            com_wr('title off.', COLORMESS);
        end;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if (s = 'on') or (s = 'off') then
  begin { Special: turn on/off titles. }
    if s = 'on' then
      temp := 1
    else
      temp := 0;
    UpdateVitals;
```

```
      for i := 1 to sc.num do
        with sc.ls[i] do
          if sel = 1 then
          begin
            ti.on := temp;
            sc_resize(i);
            Update(i);
          end;
      exit;
    end
    else
      if sc.sel = 1 then
      begin
        for i := 1 to sc.num do
          with sc.ls[i] do
            if sel = 1 then
            begin
              if ti.on = 1 then
              begin
                ti.s := s;
                DrawScreen(i);
              end
              else
              begin
                str(i, s2);
                com_wr('Screen ' + s2 + ' title off!', COLORHL);
              end;
            end;
        DrawVitals;
      end
  end
  else if s = 'x' then
  begin
    temp := com.cur;
    for i := 1 to sc.num do
      if sc.ls[i].sel = 1 then
      begin
        do_x(i);
        com.cur := temp;
      end;
  end
  else if s = 'y' then
  begin
    temp := com.cur;
    for i := 1 to sc.num do
      if sc.ls[i].sel = 1 then
      begin
        do_y(i);
        com.cur := temp;
      end;
  end
  else
    com_err;
end;

procedure do_sc_ss(w : integer);
{ Split screen w into subscreens. }
var
  dummy : integer; { For val. }
  ix, iy : integer; { Generic counters. }
  numx, numy : integer; { # of screens in x & y directions. }
  stepx, stepy : integer; { Step size in each direction. }
  scr : integer; { Screen counter. }
  temp : bufstring;
begin
  if com.cur + 2 <> com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], numx, dummy);
  inc(com.cur);
  val(com.ls[com.cur], numy, dummy);
  if (numx < 1) or (numy < 1) or (sc.num + numx * numy - 1 > sc_MAX)
    then { -1 is to account for current screen, which is resized. }
  begin
    str(sc_MAX, temp);
    com_wr('Maximum screens = ' + temp, COLORHL);
    exit;
  end;
  scr := w;
  stepx := (sc.ls[w].gr.bdy[2].x - sc.ls[w].gr.bdy[1].x) div numx;
  stepy := (sc.ls[w].gr.bdy[2].y - sc.ls[w].gr.bdy[1].y) div numy;
  for iy := 0 to numy - 1 do
    for ix := 0 to numx - 1 do
    begin
      with sc.ls[scr].gr do
      begin
        bdy[1].x := sc.ls[w].gr.bdy[1].x + ix * stepx;
        bdy[2].x := bdy[1].x + stepx;
        bdy[1].y := sc.ls[w].gr.bdy[1].y + iy * stepy;
        bdy[2].y := bdy[1].y + stepy;
      end;
      if scr > sc.num then
        sc_init(scr); { Only initialize new screens. }
      sc_resize(scr);
      if scr = w then
        scr := sc.num + 1
      else
        inc(scr);
    end;
  UpdateVitals;
  Update(w);
  w := sc.num + 1;
  inc(sc.num, numx * numy - 1);
  for w := w to sc.num do
    Update(w);
{  if sc.cur > sc.num then
    sc.cur := 1; { If prior current screen was erased, reset current screen
    to 1. }
{  for w := 1 to wv.num do { Reassign waves to legitimate screen. }
{     if wv.ls[w]^.screen > sc.num then
      wv.ls[w]^.screen := sc.cur;
  UpdateAll; }
end;

procedure do_sech;
{ If cursor on COR wave, creates new COR wave with a sech^2 fit of the
  current wave, based on vertical limits, center and fwhm. }
var
  i : integer;
  yscale : real;
begin
  if (wv.num = 0) or (wv.ls[wv.cur]^.par.dt <> DT_COR) or (wv.num =
    maxwaves) then
```

```
    exit;
  info_edges(wv.cur, 1); { Calculate old wave's center & fwhm. }
  yscale := info_ht(wv.cur); { Calculate height to scale by. }
  CreateWave(0, wv.cur); { Create new wave. }
  for i := 1 to wv.ls[wv.cur]^.par.pt_gl do
    wv.ls[wv.cur]^.data[i] := sech_sq(ptou(wv.cur, i, sc.ls[sc.cur]
      .gr.xaxismode) - info.ctr, info.fwhm) * yscale;
  info_wr(wv.cur, info.fwhm, 'fwhm', 1);
  UpdateVitals;
  Update(sc.cur);
end;

procedure do_sh;
{ Select type of wave information to show. }
var
  i : integer;
  s : bufstring;
  sh_type : integer;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  sh_type := 0;
  for i := 1 to wv_sh_MAX do
    if s = wv_sh_NAME[i] then
      sh_type := i;
  if sh_type = 0 then
  begin
    com_err;
    exit;
  end;
  wv_sel(1);
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
      begin
        if (sh_type = wv_sh_DLY) and (par.dt <> DT_ELE) then
          com_wr('Wave ' + makestringint(i) + ' cannot show DLY.',
            COLORHL)
        else
          par.sh := sh_type;
      end;
  wv_sel_off;
  DrawWaveData;
end;

procedure do_sm;
{ Smooth waves. }
const
  NO_WIDTH = 'No width given. See sm in help menu.';
var
  dummy : integer; { For val. }
  en : boolean; { Flag to save energy file (true) or regular (false). }
  r : real; { Width. }
  s : bufstring;
  i : integer; { Wave counter. }
begin
  en := false;
  { See if at end of command line; if so, signal error (no width). }
  if com.cur = com.num then
  begin
    com_wr(NO_WIDTH, COLORHL);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'en' then
  begin
    en := true;
    if com.cur = com.num then
    begin
      com_wr(NO_WIDTH, COLORHL);
      exit;
    end;
    inc(com.cur);
    s := com.ls[com.cur];
  end;
  { Read width. }
  val(s, r, dummy);
  if r <= 0 then
  begin
    com_wr('Width must be positive!', COLORHL);
    exit;
  end;
  if en and (r > 1) then
  begin
    com_wr('Width cannot exceed 1 eV!', COLORHL);
    exit;
  end;
  wv_sel(1); { Read list. }
  sc_sel_off;
  if wv.sel = 1 then
  begin
    for i := 1 to wv.num do
      with wv.ls[i]^ do
        if sel = 1 then
        begin
          s := 'Wave ' + makestringint(i) + ' ';
          if (scan.mode > 0) or ((addwaves.mode = 1) and (addwaves.w =
            i)) then
            com_wr(s + 'active! No smooth.', COLORHL)
          else if en and (par.dt <> dt_ELE) then
            com_wr(s + 'not ELE datatype! No smooth.', COLORHL)
          else
          begin
            com_wr(s + 'smoothing...', COLORMESS);
            if en then
              SmoothEnergy(i, r)
            else
              SmoothTime(i, r);
            datasaved := 0;
            sc.ls[screen].sel := 1; { Mark screen for update. }
          end;
        end;
    wv_sel_off;
    UpdateVitals;
    for i := 1 to sc.num do { Update affected screens. }
      if sc.ls[i].sel = 1 then
        Update(i);
    sc_sel_off;
  end;
end;
```

```
procedure do_ss;
{ Create split screen. See also sc ss. }
var
  dummy : integer; { For val. }
  ix, iy : integer; { Generic counters. }
  numx, numy : integer; { # of screens in x & y directions. }
  stepx, stepy : integer; { Step size in each direction. }
  scr : integer; { Screen counter. }
  temp : bufstring;
  w : integer; { Wave counter. }
begin
  if com.cur + 2 <> com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], numx, dummy);
  inc(com.cur);
  val(com.ls[com.cur], numy, dummy);
  if (numx < 1) or (numy < 1) or (numx * numy > sc_MAX) then
  begin
    str(sc_MAX, temp);
    com_wr('Maximum screens = ' + temp, COLORHL);
    exit;
  end;
  scr := 1;
  stepx := (sc.bdy[2].x - sc.bdy[1].x) div numx;
  stepy := (sc.bdy[2].y - sc.bdy[1].y) div numy;
  for iy := 0 to numy - 1 do
    for ix := 0 to numx - 1 do
    begin
      with sc.ls[scr].gr do
      begin
        bdy[1].x := sc.bdy[1].x + ix * stepx;
        bdy[2].x := bdy[1].x + stepx;
        bdy[1].y := sc.bdy[1].y + iy * stepy;
        bdy[2].y := bdy[1].y + stepy;
      end;
      if scr > sc.num then
        sc_init(scr); { Only initialize new screens. }
{      com_wr('about to SC-RESIZE', COLORDEBUG);
      if readkey = extended then readkey; }
      sc_resize(scr);
      inc(scr);
    end;
  sc.num := numx * numy;
  if sc.cur > sc.num then
    sc.cur := 1; { If prior current screen was erased, reset current screen
    to 1. }
  for w := 1 to wv.num do { Reassign waves to legitimate screen. }
    if wv.ls[w]^.screen > sc.num then
      wv.ls[w]^.screen := sc.cur;
  if osc.scr > sc.num then { Reassign osc screen. }
    osc.scr := sc.cur;
  UpdateAll;
end;

procedure do_stop;
{ Stop scanning one or more waves. }
var
  i : integer;
begin
  wv_sel(1);
  sc_sel_off;
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
      begin
        ScanStop(i);
        sc.ls[screen].sel := 1;
      end;
  wv_sel_off;
  UpdateSel;
end;

procedure do_sts;
{ Print or change STS (shot-to-shot) mode status. }

  function do_sts_err : boolean;
  { Print error message if waves scanning. }
  begin
    if bs_on and scanning(dt_ele) then
    begin
      com_wr('bs ele waves active! No change.', COLORHL);
      do_sts_err := true;
    end
    else
      do_sts_err := false;
  end;

var
  dummy : integer;
  i : integer;
  r : real;
  s : bufstring;
begin
  if (com.cur = com.num) and (not do_sts_err) then
    _bs.sts := 1
  else
  begin
    inc(com.cur);
    s := com.ls[com.cur];
    if s = '?' then
    begin
      com_wr('sts ' + makestringint(_bs.sts), COLORMESS);
      exit;
    end
    else if s = 'all' then
    begin
      com_wr('sts ' + makestringint(_bs.sts), COLORMESS);
      com_wr('sts blank ' + makestringint(_bs.sts_blank), COLORMESS);
      com_wr('sts blank2 ' + makestringint(_bs.sts_blank2),
        COLORMESS);
      com_wr('sts ch ' + chr(ord('0') + _bs.sts_ch), COLORMESS);
      com_wr('sts fac ' + makestring(_bs.sts_fac, VALMAX, VALDEC)
        + ' V/div', COLORMESS);
      com_wr('Press any key to continue.', COLORHL);
      readkey;
      com_wr('sts vert ' + makestring(_bs.sts_vert, VALMAX,
        VALDEC) + ' V/div', COLORMESS);
      com_wr('sts tog ' + chr(ord('0') + _bs.sts_tog), COLORMESS);
    end
    else if s = 'blank' then
    begin
```

```
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
    com_wr('sts blank ' + makestringint(_bs.sts_blank), COLORMESS)
  else
  begin
    val(s, i, dummy);
    if (i > 0) and (i < MAXPOINTS) then
      _bs.sts_blank := i
    else
      com_err;
  end;
end
else if s = 'blank2' then
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
    com_wr('sts blank2 ' + makestringint(_bs.sts_blank2),
      COLORMESS)
  else
  begin
    val(s, i, dummy);
    if (i > 0) and (i <= MAXPOINTS) then
      _bs.sts_blank2 := i
    else
      com_err;
  end;
end
else if s = 'ch' then
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
    com_wr('sts ch ' + chr(ord('0') + _bs.sts_ch), COLORMESS)
  else
  begin
    i := ord(s[1]) - ord('0');
    if (i > 0) and (i <= MAS_CH_MAX) then
      _bs.sts_ch := i
    else
      com_err;
  end;
end
else if s = 'fac' then
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
    com_wr('sts fac ' + makestring(_bs.sts_fac, VALMAX, VALDEC)
      + ' V/div', COLORMESS)
  else
  begin
    val(s, r, i);
    if r <= 0 then
      com_err
    else
      _bs.sts_fac := r;
  end;
end
else if s = 'vert' then
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
    com_wr('sts vert ' + makestring(_bs.sts_vert, VALMAX,
      VALDEC) + ' V/div', COLORMESS)
  else
  begin
    val(s, r, i);
    for i := TEK_VPERDIV_MIN to TEK_VPERDIV_MAX do
      if r <= TEK_VPERDIV[i] then
      begin
        _bs.sts_vert := TEK_VPERDIV[i];
        com_wr('sts vert ' + makestring(_bs.sts_vert, VALMAX,
          VALDEC) + ' V/div', COLORMESS);
        exit;
      end;
    com_err;
  end;
end
else if s = 'tog' then
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
    com_wr('sts tog ' + chr(ord('0') + _bs.sts_tog), COLORMESS)
  else
  begin
    i := ord(s[1]) - ord('0');
    if (i = 0) or (i = 1) then
      _bs.sts_tog := i
    else
      com_err;
  end;
end
```

```
    else if not do_sts_err then
    begin
      if (s = '0') or (s = 'off') then
        _bs.sts := 0
      else if (s = '1') or (s = 'on') then
        _bs.sts := 1
      else
      begin
        com_err;
        exit;
      end;
    end;
  end;
  DrawWaveData;
end;

procedure do_sv;
{ Save waves in normal or energy format; also branch out for ws, fpes
  saves. }
var
  s : bufstring;
  w : integer;
begin
  wv_sel(1);
  if com.cur < com.num then
  begin
    inc(com.cur);
    s := com.ls[com.cur];
    if s = 'en' then
    begin { Save energy format. }
      wv_sel_off; { Erase any erroneous waves from first wv_sel. }
      wv_sel(1);
      ynaesc_response := 0;
      if wv.sel = 1 then
        for w := 1 to wv.num do
          if wv.ls[w]^.sel = 1 then
            SaveEnergy(w);
    end
    else if s = 'fpes' then
    begin
      if com.cur = com.num then
      begin
        com_wr('Must supply a filename to save to.', COLORHL);
        exit;
      end;
      inc(com.cur);
      fpes_sv(com.ls[com.cur]);
      exit;
    end
    else if s = 'time' then { time-space version of data, without
      headers. }
    begin
      wv_sel_off; { Erase any erroneous waves from first wv_sel. }
      wv_sel(1);
      ynaesc_response := 0;
      if wv.sel = 1 then
        for w := 1 to wv.num do
          if wv.ls[w]^.sel = 1 then
            SaveTime(w);
    end
    else if s = 'ws' then { Special code: save worksheet with standard
      filename or user-supplied name. Note that directory dir does not
      apply to worksheet filename. }
    begin
      if wv.sel = 1 then
        wv.ls[wv.cur]^.sel := 0; { Deselect wave selected by wv_sel. }
      if com.cur = com.num then
        ws_sv(WS_FN_DF)
      else
        ws_sv(com.ls[com.cur + 1]);
      exit;
    end
    else
    begin
      com_err;
      exit;
    end;
  end
  else { Normal format. }
  begin
    ynaesc_response := 0;
    if wv.sel = 1 then
      for w := 1 to wv.num do
        if wv.ls[w]^.sel = 1 then
          SaveWave(w);
  end;
  wv_sel_off;
  DrawWaveData; { Update waves with save symbol. }
end;

procedure do_t0;
{ Set default t0 for default or active waves. }
var
  i : integer;
  s : bufstring;
  pos : real;

  procedure do_t0_change(w : integer; pos : real);
  { Change wave w's t0 to pos, or print if pos < 0. }
  begin
    with wv.ls[w]^ do
    begin
      case par.dt of
      DT_COR :
        if pos < 0 then { Flag to print current. }
          pos := par.cor.ts.t0
        else
        begin
          par.cor.ts.t0 := pos;
          UpdateCORLimits(@par); { Change start & stop. }
        end;
      DT_ELE :
        if pos < 0 then { Flag to print current. }
          pos := par.ele.ts.t0
        else
        begin
          par.ele.ts.t0 := pos;
          { Update other parameters. }
          par.ele.ts.pos := par.ele.ts.t0 + par.ele.dly *
            HALFSPEEDOFLIGHT;
          roundoff(par.ele.ts.pos, ts.step);
          limit(par.ele.ts.pos, STAGEMIN, STAGEMAX);
          par.ele.dly := (par.ele.ts.pos - par.ele.ts.t0) /
            HALFSPEEDOFLIGHT;
        end;
      else
```

```
        com_wr('Wave ' + makestringint(w) + ' not COR or ELE wave!',
          COLORHL);
      end;
      { Print out position if correct wave type, mark screen for
        update. }
      if (par.dt = DT_COR) or (par.dt = DT_ELE) then
      begin
        str(pos / POWTS : 0 : 0, s);
        com_wr('wv ' + makestringint(w) + ' t0 ' + s + ' um',
          COLORMESS);
        sc.ls[screen].sel := 1;
      end;
    end;
  end;

  procedure do_t0_get;
  { Read in t0 value. }
  var
    dummy : integer;
    s : bufstring;
  begin
    if com.cur = com.num then
    begin
      pos := ts.pos;
      exit;
    end;
    inc(com.cur);
    s := com.ls[com.cur];
    if s = '?' then
      pos := -1
    else
    begin
      val(s, pos, dummy);
      pos := pos * POWTS;
      limit(pos, STAGEMIN, STAGEMAX);
    end;
  end;

begin
  sc_sel_off;
  if com.cur = com.num then
  begin
    if wv.num = 0 then
    begin
      com_err;
      exit;
    end;
    sc.ls[wv.ls[wv.cur]^.screen].sel := 1;
    do_t0_change(wv.cur, ts.pos);
    UpdateSel;
    exit;
  end;

  inc(com.cur);
  s := com.ls[com.cur];

  if s = 'ac' then { Change/print ac default t0. }
  begin
    do_t0_get;
    if pos >= 0 then { pos < 0 is flag to display current. }
      t0.ac := pos;
    str(t0.ac / POWTS : 0 : 0, s);
    com_wr('t0 ac ' + s + ' um', COLORMESS);
    exit;
  end
  else if s = 'cc' then { Change/print cc default t0. }
  begin
    do_t0_get;
    if pos >= 0 then { pos < 0 is flag to display current. }
      t0.cc := pos;
    str(t0.cc / POWTS : 0 : 0, s);
    com_wr('t0 cc ' + s + ' um', COLORMESS);
    exit;
  end
  else if s = 'cor' then { Change/print cor default t0. }
  begin
    do_t0_get;
    if pos >= 0 then { pos < 0 is flag to display current. }
    begin
      pardf[dt_COR].cor.ts.t0 := pos;
      UpdateCORLimits(@pardf[dt_COR]); { Change start & stop. }
    end;
    str(pardf[dt_COR].cor.ts.t0 / POWTS : 0 : 0, s);
    com_wr('t0 cor ' + s +  'um', COLORMESS);
    exit;
  end
  else if s = 'ele' then { Change/print ele default t0. }
  begin
    do_t0_get;
    if pos >= 0 then { pos < 0 is flag to display current. }
      with pardf[dt_ELE] do
      begin
        ele.ts.t0 := pos;
        { Update other parameters. }
        ele.ts.pos := ele.ts.t0 + ele.dly * HALFSPEEDOFLIGHT;
        roundoff(ele.ts.pos, ts.step);
        limit(ele.ts.pos, STAGEMIN, STAGEMAX);
        ele.dly := (ele.ts.pos - ele.ts.t0) / HALFSPEEDOFLIGHT;
      end;
    str(pardf[dt_ELE].ele.ts.t0 / POWTS : 0 : 0, s);
    com_wr('t0 ele ' + s + ' um', COLORMESS);
    exit;
  end;

  if s = 'cur' then { Change waves to current position. }
    pos := ts.pos
  else if s = '?' then { Print current position of waves. }
    pos := -1
  else { Assume entry is a number. }
  begin
    dec(com.cur);
    do_t0_get;
  end;

  wv_sel(1); { Get list of waves. }

  { Now assign pos to appropriate waves. }
  for i := 1 to wv.num do
    if wv.ls[i]^.sel = 1 then
      do_t0_change(i, pos);

  { Update screen. }
  wv_sel_off;
  UpdateSel;
end;
```

```
procedure do_ts;
{ Handle translation stage commands. }
const
  LF = #10; { Compiler seems to need this declaration here rather than in the
    Keys unit. }
var
  del : longint;
  dummy : integer;
  exitflag : boolean;
  r : real;
  s, s2 : bufstring;
  w : integer;
begin
  if com.cur = com.num then
  begin { Print current stage position. }
    str(ts.pos / POWTS : 0 : VALDECTS, s);
    com_wr('Position ' + s + ' um', COLORMESS);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'acc' then
  begin
    if com.cur = com.num then
    begin
      com_wr('ts acc ' + makestringint(ts.acc), COLORMESS);
      exit;
    end;
    inc(com.cur);
    val(com.ls[com.cur], ts.acc, dummy);
    if ts.acc <= 0 then
      ts.acc := TS_ACC_DF;
    write(COM2, ACK);
    delay(ACKDELAY);
    write(COM2, '#FCAB6' + LF + 'B' + makestringint(ts.acc) + LF);
    ts.wait.int := 100 * ts.vel div ts.acc;
    if ts.wait.int < TINT_DF then
      ts.wait.int := TINT_DF;
  end
  else if s = 'ack' then
  begin
    write(COM2, ACK);
    delay(ACKDELAY);
  end
  else if s = 'ampl' then { Print/change amplitude of wob function. }
  begin
    if com.cur = com.num then
    begin
      com_wr('ts.wob.ampl ' + makestring(ts.wob.ampl / POWTS,
        VALMAX, VALDEC) + ' um.', COLORMESS);
      exit;
    end;
    inc(com.cur);
    val(com.ls[com.cur], ts.wob.ampl, dummy);
    ts.wob.ampl := ts.wob.ampl * POWTS;
    MakeLookup; { Update lookup table. }
  end
  else if s = 'com' then
  begin
    if com.cur = com.num then
    begin { Must be a command after 'com'. }
      com_err;
      exit;
    end;
    inc(com.cur);
    s := com.ls[com.cur];
    write(COM2, ACK);
    delay(ACKDELAY);
    { Convert linefeed symbol '\' to real linefeed code. }
    for w := 1 to length(s) do
      if s[w] = '\' then
        s[w] := LF;
    { See if #CA or non-#CA command. }
    if s[1] = '#' then
      write(COM2, s + LF)
    else
      write(COM2, '#CA' + s + LF);
  end
  else if s = 'home' then
  begin
    write(COM2, ACK);
    delay(ACKDELAY);
    write(COM2, '#CAHM' + LF);
    com_wr(com_wr_MOVINGSTAGE, COLORHL);
    if readkey = EXTENDED then
      readkey;
    write(COM2, ACK);
    delay(ACKDELAY);
    write(COM2, '#CAABSL' + LF);
    ts.pos := 0;
  end
  else if s = 'init' then
  begin
    { Send setup commands. }
    com_wr('Initializing stage...', COLORMESS);
    write(COM2, #3);
    DELAY(1000);
    write(COM2, #32);
    DELAY(1000);
    write(COM2, #30);
    DELAY(1000);
    write(COM2, ACK);
    DELAY(ACKDELAY);
    { Home stage. }
{   com_wr(com_wr_MOVINGSTAGE, COLORHL);
    write(COM2, '#CAHM' + LF);
    if readkey = EXTENDED then
      readkey;
    write(COM2, ACK);
    { Set absolute mode. }
    write(COM2, '#CAABSL' + LF);
    delay(1000);
    write(COM2, ACK);
    { Set current acceleration and velocity. }
    delay(ACKDELAY);
    write(COM2, '#FCAB6' + LF + 'B' + makestringint(ts.acc) + LF);
    delay(ACKDELAY);
    write(COM2, ACK);
    delay(ACKDELAY);
    write(COM2, '#FCAB7' + LF + 'B' + makestringint(ts.vel) + LF);
{   delay(ACKDELAY);
    write(COM2, ACK); }
    ts.pos := 0;
  end
  else if s = 'int' then { Minimum time constant. }
  begin
```

```
  if com.cur = com.num then
    com_wr('ts.wait.int ' + makestringint(ts.wait.int), COLORMESS)
  else
  begin
    inc(com.cur);
    val(com.ls[com.cur], ts.wait.int, dummy);
    if ts.wait.int < 0 then
      ts.wait.int := TINT_DF;
  end;
end
else if s = 'per' then { Print/change period of wob function. }
begin
  if com.cur = com.num then
  begin
    com_wr('ts.wob.per ' + makestring(ts.wob.per / POWTS,
      VALMAX, VALDEC) + ' um.', COLORMESS);
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], ts.wob.per, dummy);
  ts.wob.per := ts.wob.per * POWTS;
  MakeLookup; { Update lookup table. }
end
else if s = 'ph' then { Print/change phase of wob function. }
begin
  if com.cur = com.num then
  begin
    com_wr('ts.wob.ph ' + makestring(ts.wob.ph,
      VALMAX, VALDEC), COLORMESS);
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], ts.wob.ph, dummy);
  ts.wob.ph := frac(1 + frac(ts.wob.ph)); { Keep number
    between 0 and 1 only. }
  { No need to call MakeLookup -- phase not used there. }
end
else if s = 'rd' then
begin
  { Reopen file for reading. }
  close(COM2);
  assign(COM2, 'COM2');
  reset(COM2);
  { Read characters until user hits CR. }
  com_wr('Press any key to read next char, or RETURN to exit.',
    COLORMESS);
  while readkey <> CR do
  begin
    read(COM2, s[1]);
    com_wr('ts char = ' + makestringint(ord(s[1])), COLORMESS);
  end;
  { Reopen file for writing again. }
  close(COM2);
  assign(COM2, 'COM2');
  rewrite(COM2);
end
else if s = 'sl' then { Slope time constant. }
begin
  if com.cur = com.num then
    com_wr('ts.wait.sl ' + makestringint(ts.wait.sl), COLORMESS)
  else
  begin
    inc(com.cur);
    val(com.ls[com.cur], ts.wait.sl, dummy);
    if ts.wait.sl <= 0 then
      ts.wait.sl := TSL_DF;
  end;
end
else if s = 'step' then { Stepsize. }
begin
  if com.cur = com.num then
  begin
    com_wr('ts.step ' + makestring(ts.step / POWTS, VALMAXTS,
      VALDECTS) + ' um', COLORMESS);
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], ts.step, dummy);
  ts.step := ts.step * POWTS;
  if ts.step < TS_STEP_MIN then
    ts.step := TS_STEP_MIN;
  write(COM2, ACK);
  delay(ACKDELAY);
  write(COM2, '#CAPRM:203=' + makestringint(round(TS_RES_STEP /
    ts.step)) + LF);
end
{  else if s = 'step' then
  begin
    if com.cur = com.num then
    begin { Print current step. }
{     str(move.step / POWTS : 0 : 0, s);
      com_wr('Step = ' + s + ' um', COLORMESS);
      exit;
    end;
    { Read new step. }
{    inc(com.cur);
    val(com.ls[com.cur], move.step, dummy);
    move.step := move.step * POWTS;
    roundoff(move.step, ts.step);
    limit(move.step, ts.step, STAGEMAX);
  end }
  else if s = 'tog' then { Toggle. }
  begin
    { Use current values. }
    if com.cur = com.num then
    begin
      do_ts_tog;
      exit;
    end;
    inc(com.cur);
    s := com.ls[com.cur];
    if s = 'dly' then
    begin { Print or change delay time. }
      do_ts_dly;
      exit;
    end;
    { Check that at least one more entry in command (must provide both
      positions!) }
    if com.cur = com.num then
    begin
      com_err;
      exit;
    end;
    { Read positions. }
    val(s, move.start, dummy);
    move.start := move.start * POWTS;
```

```
  roundoff(move.start, ts.step);
  limit(move.start, STAGEMIN, STAGEMAX);
  inc(com.cur);
  val(com.ls[com.cur], move.stop, dummy);
  move.stop := move.stop * POWTS;
  roundoff(move.stop, ts.step);
  limit(move.stop, STAGEMIN, STAGEMAX);
  { Recalculate stage delay. }
  r := ts.pos; { Save current stage position. }
  ts.pos := move.start;
  move.wait := 10 * abs(StageDelay(move.stop)); { Ignore user input
    flag. }
  ts.pos := r; { Restore. }
  { Check if 'dly' occurs next (can also change delay). }
  if com.cur < com.num then
  begin
    inc(com.cur);
    if com.ls[com.cur] = 'dly' then
      do_ts_dly;
  end;
  { Do toggle. }
  do_ts_tog;
end
else if s = 'vel' then
begin
  if com.cur = com.num then
  begin
    com_wr('ts vel ' + makestringint(ts.vel), COLORMESS);
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], ts.vel, dummy);
  if ts.vel <= 0 then
    ts.vel := TS_VEL_DF
  else if ts.vel >= TS_VEL_MAX then
    ts.vel := TS_VEL_MAX;
  write(COM2, ACK);
  delay(ACKDELAY);
  write(COM2, '#FCAB7' + LF + 'B' + makestringint(ts.vel) + LF);
  ts.wait.sl := 100000000 div ts.vel;
end
else if s = 'walk' then
begin
  { Make sure not interfering with active waves. }
  if scanning(dt_COR) or scanning(dt_ELE)
    then
  begin
    if com_wr_yn('Stage in use. Stop active waves') = 0 then
      exit
    else
    begin
      for w := 1 to wv.num do
        with wv.ls[w]^ do
          if (scan.mode > 0) and (par.dt <> dt_POW) then
            ScanStop(w);
      UpdateAll;
    end;
  end;
  exitflag := FALSE;
  { Main loop. }
  repeat
    str(ts.pos / POWTS : 0 : VALDECTS, s);
    str(move.step / POWTS : 0 : VALDECTS, s2);
    com_wr('Pos ' + s + ' Step ' + s2 + ' um', COLORMESS);
    case readkey of
    EXTENDED:
      case readkey of
      XARROWUP:
        begin
          move.step := move.step * 2;
          if move.step > STAGEMAX then
            move.step := STAGEMAX;
        end;
      XARROWDOWN: { Down arrow }
        if move.step > ts.step then
          move.step := int((move.step / ts.step)) * ts.step / 2;
      XARROWLEFT: { Left arrow }
        if ts.pos - move.step >= STAGEMIN then
          StageMoveWait(ts.pos - move.step);
      XARROWRIGHT: { Right arrow }
        if ts.pos + move.step <= STAGEMAX then
          StageMoveWait(ts.pos + move.step);
      end; { Extended keys. }
    ESC: { Exit. }
      begin
        write(COM2, ACK);
        exitflag := TRUE;
      end;
    end;
  until exitflag = TRUE;
end
else if s = 'wob' then { Print all 3 parameters. }
begin
  com_wr('ts.wob.ampl ' + makestring(ts.wob.ampl / POWTS,
    VALMAX, VALDEC) + ' um.', COLORMESS);
  com_wr('ts.wob.per ' + makestring(ts.wob.per / POWTS,
    VALMAX, VALDEC) + ' um.', COLORMESS);
  com_wr('ts.wob.ph ' + makestring(ts.wob.ph,
    VALMAX, VALDEC), COLORMESS);
end
else if s = 'x' then { Disengage. }
begin
  write(COM2, ACK);
  delay(ACKDELAY);
  write(COM2, '#EA' + LF);
end
else
begin { Move stage. }
  val(s, r, dummy);
  r := r * POWTS;
  roundoff(r, ts.step);
  if (r < STAGEMIN) or (r > STAGEMAX) then
  begin
    com_err;
    exit;
  end;
  { Check that no ts-dependent waves scanning. }
  if scanning(dt_COR) or scanning(dt_ELE)
    then
  begin
    if com_wr_yn('Stage in use. Stop active waves') = 0 then
      exit
    else
    begin
      for w := 1 to wv.num do
        with wv.ls[w]^ do
```

```
            if (scan.mode > 0) and (par.dt <> dt_POW) then
              ScanStop(w);
          UpdateAll;
        end;
      end;
      StageMoveWait(r);
    end;
end;

procedure do_ts_dly;
{ Handle dly. }
var
  dummy : integer;
  s : bufstring;
begin
  if com.cur = com.num then
  begin { Print current delay. }
    str(round(move.wait), s);
    com_wr('move.wait = ' + s + ' ms', COLORMESS);
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], move.wait, dummy);
end;

procedure do_ts_tog;
{ Toggle between stage positions move.start and move.stop, waiting for
  move.wait ms each move. Stops when user hits a key. }
var
  del : longint; { Delay. }
  s, s2 : bufstring;
  sum : real;
  w : integer;
begin
  { Ensure we don't interfere with active waves. }
  if scanning(dt_COR) or scanning(dt_ELE)
    then
  begin
    if com_wr_yn('Stage in use. Stop active waves') = 0 then
      exit
    else
    begin
      for w := 1 to wv.num do
        with wv.ls[w]^ do
          if (scan.mode > 0) and (par.dt <> dt_POW) then
            ScanStop(w);
      UpdateAll;
    end;
  end;
  sum := move.start + move.stop; { Math trick: add limits together; then
    can toggle between the values by subtracting ts.pos from it each
    time (see below). }
  { Move to starting position. }
  StageMoveWait(move.start);
  com_wr('Toggle: ' + makestringint(move.start / POWTS) + ', ' +
    makestringint(move.stop / POWTS) + ' um', COLORMESS);
  com_wr('Press any key to exit.', COLORMESS);
  { Main loop. }
  repeat
    StageMove(sum - ts.pos);
    write(BELL);
    delay(round(move.wait));
  until keypressed;
  { Remove character from buffer: }
  if readkey = EXTENDED then
    readkey;
end;

procedure do_tw;
{ Tweak mode: Allow fast MCS scanning to find weak electron signals. }
var
  dummy : integer;
  p : real;
  s : bufstring;
  t : longint; { Timekeeping variable. }
begin
  if com.cur < com.num then
  begin
    inc(com.cur);
    s := com.ls[com.cur];
    p := power(tw.timeperpt);
    if s = 'all' then
    begin
      com_wr('tw.reprate ' + makestringint(tw.reprate) + ' Hz',
        COLORMESS);
      com_wr('tw.shotsperscan ' + makestringint(tw.shotsperscan),
        COLORMESS);
      com_wr('tw.timeperpt ' + makestring(tw.timeperpt / p, VALMAX,
        0) + ' ' + UnitPrefix(round(log10(p))) + 's', COLORMESS);
    end
    else if s = 'rep' then
    begin
      if com.cur = com.num then
        com_wr('tw.reprate ' + makestringint(tw.reprate) + ' Hz',
          COLORMESS)
      else
      begin
        val(com.ls[com.cur + 1], tw.reprate, dummy);
        if tw.reprate < 1 then
          tw.reprate := 1;
      end;
    end
    else if s = 'shots' then
    begin
      if com.cur = com.num then
        com_wr('tw.shotsperscan ' + makestringint(tw.shotsperscan),
          COLORMESS)
      else
      begin
        val(com.ls[com.cur + 1], tw.shotsperscan, dummy);
        if tw.shotsperscan < 1 then
          tw.shotsperscan := 1
        else if tw.shotsperscan > MAXSHOTSPERSCAN then
          tw.shotsperscan := MAXSHOTSPERSCAN;
      end;
    end
    else if s = 'time' then
    begin
      if com.cur = com.num then
        com_wr('tw.timeperpt ' + makestring(tw.timeperpt / p, VALMAX,
          0) + ' ' + UnitPrefix(round(log10(p))) + 's', COLORMESS)
      else
      begin
        val(com.ls[com.cur + 1], tw.timeperpt, dummy);
        tw.timeperpt := MCS_TIMEPERPT[TimebaseToCode(tw.timeperpt *
          p)]; { Force to match acceptable value. }
```

```
      end;
    end
    else
      com_err;
    exit;
  end;

  { Invalidate currently scanning wave if ele datatype (will resume
    after tweak has ended): }
  if scanwave > 0 then
    with wv.ls[scanwave]^ do
      if (par.dt = DT_ELE) and (scan.mode > 0) then
        scan.mode := SCAN_MODE_NEW;

  com_wr('Tweak using MCS. Press any key to exit.', COLORMESS);
  mcs.new := 1;
  { Main loop. }
  repeat
    { Start scan. }
    if MCS_init(0) = false then
      exit;
    { Now wait for user keypress or time to elapse. }
    repeat
      time(t);
      if (t < tw.starttime) then { Handle wraparound. }
        dec(tw.starttime, TIMEMAX - tw.steptime);
    until (t - tw.starttime > tw.steptime) or keypressed;
    delay(250);
{    mwrite(mcs.addr, 'stat', false);
    mwrite(mcs.addr, 'spar? 2', false); { Read total area. }
{    mread(mcs.addr, 20);
    com_wr('Total counts ' + rdbuf, COLORMESS); }
{    mwrite(mcs.addr, 'scan?', false);
    delay(500);
    mread(mcs.addr, 10);
    Val(rdbuf, scan.shots, result);
    if scan.shots < par.ele.shotsperscan then
      exit; }
  until keypressed;

  { Remove pressed key from buffer: }
  if readkey = EXTENDED then
    readkey;

  { Send full command set next time. }
  mcs.new := 1;
end;

procedure do_vis(v : integer);
{ Turn on/off selected waves according to v. Command may be chained,
  i.e. 'vis 1 2 3 inv 4 5 6 vis 7 8 9' etc. }
var
  i : integer;
  s : bufstring;
begin
  { Get wave list. }
  wv_sel(1);
  { See if at end of command line; otherwise, signal error (unknown
    word). }
{  if com.cur < com.num then
  begin
    inc(com.cur); { Put cursor on mystery word. }
{    com_err;
    exit;
  end; }
  if wv.sel = 1 then
  begin
    sc_sel_off;
    { Turn on/off waves and also mark screens affected. Reset tags. }
    for i := 1 to wv.num do
      with wv.ls[i]^ do
        if sel = 1 then
          with sc.ls[screen] do
          begin
            sel := 1;
            on := v;
            { Force xaxis datatype to be same as latest wave turned
              on. }
            if v = 1 then
              gr.xaxisdt := par.dt;
          end;
    wv_sel_off;
    UpdateSel;
  end;
  if com.cur < com.num then
  { Allow chaining of inv/vis commands. }
  begin
    inc(com.cur);
    s := com.ls[com.cur];
    if s = 'inv' then
      do_vis(0)
    else if s = 'vis' then
      do_vis(1)
    else
      com_err;
  end;
end;

procedure do_ws;
{ Only handle ws df right now. }
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  if com.ls[com.cur] = 'df' then
    ws_df
  else
    com_err;
end;

procedure do_wv;
{ Create new wave, change wave, print or change wave variables. }
var
  s : bufstring;
  w : integer; { Selected wave. }
begin
  if com.cur = com.num then
  begin
    if wv.num = 0 then
      com_wr('No waves loaded.', COLORMESS)
    else
      com_wr('Current wave ' + makestringint(wv.cur), COLORMESS);
    exit;
```

```
  end;
  wv_sel(1); { Get list of waves. }
  if wv.sel = 0 then
    exit;
  if com.cur = com.num then
    { Simply switch waves: find first selected wave and use that. }
    for w := 1 to wv.num do
      if wv.ls[w]^.sel = 1 then
      begin
        if wv.ls[wv.cur]^.screen = sc.cur then
          EraseCursor(sc.cur);
        wv.cur := w;
        for w := w to wv.num do { Turn off all selections. }
          wv.ls[w]^.sel := 0;
        ChangeCurrentWave(0); { Includes drawwavedata call. }
        exit;
      end;
  { Load up next word. }
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'alert' then
    do_wv_alert
  else if (s = 'area') or (s = 'avg') or (s = 'ctr') or (s = 'fwhm') or
    (s = 'ht') or (s = 'width') then
  begin
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
          do_wv_info(w, s);
  end
  else if s = 'bg' then { Background subtraction. }
    do_wv_bg
  else if s = 'dly' then { Delay (electron waves only). }
    do_wv_dly
  else if s = 'dots' then { Change to dots. }
    do_wv_lines(0)
  else if s = 'fn' then { Filename. }
    do_wv_fn
  else if s = 'lines' then { Change to lines. }
    do_wv_lines(1)
  else if s = 'sc' then { Screen. }
    do_wv_sc
  else if s = 'sh' then { Choose what kind of information to display in
    wavedata. }
    do_wv_sh
  else if s = 'skip' then
    do_wv_skip
  else if s = 'vstop' then
    do_wv_vstop
  else
    com_err;
  wv_sel_off;
  DrawWaveData;
end;

procedure do_wv_alert;
{ Print/change par.alert flag. }
var
  dummy : integer;
  i : integer;
  w : integer;
begin
  if com.cur = com.num then
  begin
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
          com_wr('Wave ' + makestringint(w) + ' alert ' + makestringint
            (par.alert), COLORMESS);
    wv_sel_off;
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], i, dummy);
  if (i = 0) or (i = 1) then
  begin
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
          par.alert := i;
  end
  else
    com_err;
  wv_sel_off;
end;

procedure do_wv_bg;
{ Handle wave background subtraction variables. }
var
  dummy : integer;
  i : integer;
  s, s2 : bufstring;
  temp : integer;
  w : integer;
begin
  for w := 1 to wv.num do
    with wv.ls[w]^ do
      if sel = 1 then
      begin
        temp := com.cur;
        str(w, s2);
        s2 := 'Wave ' + s2 + ' ';
        if par.dt <> dt_ELE then
          com_wr(s2 + ' not ele wave.', COLORHL)
        else
        begin
          if com.cur = com.num then { Write current background wave. }
          begin
            str(par.ele.bs.mode, s);
            com_wr(s2 + 'par.ele.bs.mode = ' + s, COLORMESS);
          end
          else
          begin
            inc(com.cur);
            s := com.ls[com.cur];
            val(s, i, dummy);
            if (i >= 1) and (i <= wv.num) then
            begin
              { Change background wave. Checks a few parameters to ensure
                compatibility. }
              with wv.ls[i]^ do
                if (par.timeperpt = wv.ls[w]^.par.timeperpt) and
                  (par.pt = wv.ls[w]^.par.pt)
                  and (par.ele.bs.mode = 0) and (i < w) then
                  wv.ls[w]^.par.ele.bs.mode := i
                else
```

```
                com_wr(s2 + 'background wave incompatible.', COLORHL);
            end
            else if s = '0' then
              par.ele.bs.mode := 0 { Turn off background mode. }
            else if s = 'scale' then
              rd_real(par.ele.bs.tot, 'background scaling', 0,
                LARGE)
            else
            begin
              com_err;
              exit;
            end;
          end;
        end;
        com.cur := temp;
      end;
end;

procedure do_wv_dly;
{ Print/change time delay of electron waves. }
var
  dummy : integer;
  r : real;
  s : bufstring;
  w : integer;
begin
  if com.cur = com.num then
  begin { Print time delay. }
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
        begin
          sel := 0;
          str(w, s);
          s := 'Wave ' + s + ' ';
          if par.dt <> dt_ELE then
            com_wr(s + 'not electron wave.', COLORHL)
          else
            com_wr(s + 'delay = ' + makestringint(par.ele.dly / POWFS)
              + ' fs', COLORMESS);
        end;
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], r, dummy);
  r := r * POWFS;
  roundoff(r, ts.step / HALFSPEEDOFLIGHT);
  for w := 1 to wv.num do
    with wv.ls[w]^ do
      if sel = 1 then
      begin
        sel := 0;
        str(w, s);
        s := 'Wave ' + s + ' ';
        { Ensure is electron wave. }
        if par.dt <> dt_ELE then
          com_wr(s + 'not electron wave.', COLORHL)
        { Ensure not scanning. }
        else if scan.mode > 0 then
          com_wr(s + 'scanning.', COLORHL)
        else
        begin
          { Calculate stage position and keep within limits. }
          par.ele.ts.pos := par.ele.ts.t0 + r * HALFSPEEDOFLIGHT;
          limit(par.ele.ts.pos, STAGEMIN, STAGEMAX);
          { Now assign time delay. }
          par.ele.dly := (par.ele.ts.pos - par.ele.ts.t0) /
            HALFSPEEDOFLIGHT;
          { Print to screen to be sure user knows. }
          com_wr(s + 'delay ' + makestringint(par.ele.dly / POWFS) +
            ' fs', COLORMESS);
        end;
      end;
  DrawWaveData; { Update info since waves may display dly. }
end;

procedure do_wv_fn;
{ Print/change filename of waves. }
var
  s : bufstring;
  w : integer;
begin
  if com.cur = com.num then
  begin { Print filenames. }
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
        begin
          sel := 0;
          str(w, s);
          com_wr('Wave ' + s + ': ' + par.fn, COLORMESS);
        end;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  for w := 1 to wv.num do
    with wv.ls[w]^ do
      if sel = 1 then
      begin
        sel := 0;
        par.fn := s;
      end;
  UpdateFilenames(dir);
  DrawWaveData; { Update info since waves may display fn. }
end;

procedure do_wv_info(w : integer; s2 : bufstring);
{ Reports information about waves, depending on s2:
  'area' : Calculates area under wave.
  'avg' : Calculates average value (area / x range).
  'ctr' : Calculates x position of "center", defined as halfway between
    the two 50% marks as seen from left and right sides of wave.
  'fwhm' : Calculates width of wave, defined as distance
    between 50% marks from left and right sides of wave.
  'ht': Calculates height of wave, defined as max - min values. }
var
  vfind : integer;
begin
  { See if vfind flag present. }
  vfind := 0;
  if com.cur < com.num then
  begin
    inc(com.cur);
    if com.ls[com.cur] = 'vfind' then
      vfind := 1
```

```
    else
      dec(com.cur); { Back up for wv_sel. }
  end;

  if w = 0 then
    wv_sel(1)
  else
  { Special handling : make only specified wave work. }
  begin
    wv_sel_off;
    wv.ls[w]^.sel := 1;
  end;
  for w := 1 to wv.num do
    if wv.ls[w]^.sel = 1 then
    begin
      if s2 = 'area' then
        info_wr(w, info_area(w), 'area', 0)
      else if s2 = 'avg' then
        info_wr(w, info_avg(w), 'average', 0)
      else if s2 = 'ht' then
        info_wr(w, info_ht(w), 'height', 0)
      else
      begin
        info_edges(w, vfind);
        if s2 = 'ctr' then
          info_wr(w, info.ctr, 'center', 1)
        else if s2 = 'edgel' then
          info_wr(w, info.edgel, 'edgel', 1)
        else if s2 = 'edger' then
          info_wr(w, info.edger, 'edger', 1)
        else if s2 = 'fwhm' then
          info_wr(w, info.fwhm, 'fwhm', 1)
        else
        begin
          com_err;
          w := wv.num; { Get out of loop. }
        end;
      end;
    end;
    wv_sel_off;
    DrawWaveData; { Clear tags. }
  end;

  procedure do_wv_lines(l : integer);
  { Change waves to lines (l = 1) or dots (l = 0). }
  var
    i : integer;
  begin
    if wv.sel = 0 then
      exit;
    for i := 1 to sc_MAX do
      sc.ls[i].sel := 0;
    for i := 1 to wv.num do
     with wv.ls[i]^ do
       if sel = 1 then
       begin
         lines := l;
         sel := 0;
         sc.ls[screen].sel := 1;
       end;
    for i := 1 to sc_MAX do
      if sc.ls[i].sel = 1 then
        DrawScreen(i);
    DrawVitals;
  end;

  procedure do_wv_sc;
  { Print/change screen of waves. }
  var
    dummy : integer;
    i : integer;
    s : bufstring;
    w : integer;
  begin
    if com.cur = com.num then { Print wave's screen. }
    begin
      for w := 1 to wv.num do
        with wv.ls[w]^ do
          if sel = 1 then
            com_wr_wv(w, 'sc ' + makestringint(screen), COLORMESS);
      exit;
    end;

    { Find screen #. }
    inc(com.cur);
    s := com.ls[com.cur];
    if s = 'df' then
      i := 0 { Flag to use default scheme. }
    else
    begin
      val(s, i, dummy);
      if (i < 1) or (i > sc.num) then
      begin
        com_err;
        exit;
      end;
    end;

    { Do screen assignment. }
    sc_sel_off;
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
        begin
          sc.ls[screen].sel := 1; { Update old screen. }
          if i = 0 then
            screen := (w - 1) mod sc.num + 1 { Cycle thru screen numbers
              using wave number. }
          else
            screen := i;
          sc.ls[screen].sel := 1; { Must update new screen too. }
          if w = wv.cur then { Change screens if current wave changes.)
            sc.cur := screen;
          sc.ls[screen].gr.xaxisdt := par.dt; { Make screen datatype
            reflect latest wave moved to it. }
        end;
    wv_sel_off;
    UpdateSel;
  end;

  procedure do_wv_sh;
  { Print/change show parameter. }
  var
    i : integer;
    s : bufstring;
    w : integer;
```

```
begin
  if com.cur = com.num then
  begin
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
        begin
          str(w, s);
          com_wr('Wave ' + s + ' par.sh = ' + wv_sh_NAME[par.sh], COLORMESS);
        end;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  i := 1;
  while (i <= wv_sh_MAX) and (s <> wv_sh_NAME[i]) do
    inc(i);
  if i <= wv_sh_MAX then
  begin
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
        begin
          if (i = wv_sh_DLY) and (par.dt <> dt_ELE)
            then
          begin
            str(w, s);
            com_wr('Wave ' + s + ' cannot show dly.', COLORHL);
          end
          else
            par.sh := i;
        end;
    DrawWaveData;
  end
  else
  begin
    com_err;
    exit;
  end;
end;

procedure do_wv_skip;
{ Print/change skip parameter. }
var
  dummy : integer;
  i : integer;
  w : integer;
begin
  if com.cur = com.num then
  begin
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
          com_wr('Wave ' + makestringint(w) + ' skip ' + makestringint
            (par.skip), COLORMESS);
    wv_sel_off;
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], i, dummy);
  if i >= 0 then
  begin
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
          par.skip := i;
  end
  else
    com_err;
  wv_sel_off;
end;

procedure do_wv_vstop;
{ Handle print/change vstop. }
var
  dummy : integer;
  i : integer;
  w : integer;
begin
  if com.cur = com.num then
  begin
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
          com_wr('Wave ' + makestringint(w) + ' vstop ' + makestringint
            (par.vstop), COLORMESS);
    wv_sel_off;
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], i, dummy);
  if i >= 0 then
  begin
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
          par.vstop := i;
  end
  else
    com_err;
  wv_sel_off;
end;


procedure do_x(scr : integer);
{ Handle x axis functions. }
var
  dummy : integer;
  s, s2 : bufstring;
  ul, u2 : real;
begin
  with sc.ls[scr].gr do
  begin
  if com.cur = com.num then
  begin { Print current x range. }
    str(scr, s);
    com_wr('Screen ' + s + ' x = ' + makestring(ullim / xpower,
      MAXXDIGITS, MAXXDIGITS - 2) + ' ' + makestring(u2lim / xpower,
      MAXXDIGITS, MAXXDIGITS - 2) + ' ' + xunits, COLORMESS);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'conv' then
    ToggleXAxisMode(scr, XAXISMODE_CONVERT)
  else if s = 'fu' then
```

```
  begin
    sc.ls[scr].gr.xfullmode := 1;
    UpdateVitals;
    Update(scr);
  end
  else if s = 'nor' then
    ToggleXAxisMode(scr, XAXISMODE_NORMAL)
  else if s = 'off' then
  begin
    xon := 0;
    sc_resize(scr);
    UpdateVitals;
    Update(scr);
  end
  else if s = 'on' then
  begin
    xon := 1;
    sc_resize(scr);
    UpdateVitals;
    Update(scr);
  end
  else if s = 'pt' then
    ToggleXAxisMode(scr, XAXISMODE_POINTS)
  else
  begin
    val(s, u1, dummy);
    if com.cur = com.num then
    begin
      com_err;
      exit;
    end;
    inc(com.cur);
    val(com.ls[com.cur], u2, dummy);
    with sc.ls[scr].gr do
    begin
      if u1 >= u2 then
      begin
        com_err;
        exit;
      end;
      xfullmode := 0;
      u1lim := u1 * xpower;
      u2lim := u2 * xpower;
    end;
    UpdateVitals;
    Update(scr);
  end;
  end;
end;

procedure do_xh;
{ Crosshairs commands. }
var
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    ToggleCrosshairsMode;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'fu' then
  begin
    with sc.ls[sc.cur].gr do
    begin
      if xh.mode = 0 then
        exit;
      EraseCursor(sc.cur);
      xh.x[1] := plotarea.x1;
      xh.x[2] := plotarea.x2 - 1;
      xh.y[1] := plotarea.y1;
      xh.y[2] := plotarea.y2 - 1;
      UpdateCursor(sc.cur);
      DrawCursor(sc.cur);
    end;
    exit;
  end
  else
    com_err;
end;

procedure do_y(scr : integer);
{ Handle y axis functions. }
var
  dummy : integer;
  s, s2 : bufstring;
  v1, v2 : real;
begin
  with sc.ls[scr].gr do
  begin
  if com.cur = com.num then
  begin { Print current y range. }
    str(scr, s);
    com_wr('Screen ' + s + ' y = ' + makestring(v1lim / ypower,
      MAXYDIGITS, MAXYDIGITS - 2) + ', ' + makestring(v2lim / ypower,
      MAXYDIGITS, MAXYDIGITS - 2) + ' ' + yunits, COLORMESS);
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = 'abs' then
  begin
    if sc.ls[scr].gr.yaxismode = YAXISMODE_RELATIVE then
      ToggleYAxisMode(scr);
  end
  else if s = 'fu' then
  begin
    sc.ls[scr].gr.yfullmode := 1;
    UpdateVitals;
    Update(scr);
  end
  else if s = 'off' then
  begin
    yon := 0;
    sc_resize(scr);
    UpdateVitals;
    Update(scr);
  end
  else if s = 'offres' then
    rd_int(yoffsetrescale, 'y offset rescale', 0, 1)
  else if s = 'on' then
  begin
    yon := 1;
    sc_resize(scr);
    UpdateVitals;
```

```
    Update(scr);
  end
  else if s = 'rel' then
  begin
    if sc.ls[scr].gr.yaxismode = YAXISMODE_ABSOLUTE then
      ToggleYAxisMode(scr);
  end
  else
  begin
    val(s, v1, dummy);
    if com.cur = com.num then
    begin
      com_err;
      exit;
    end;
    inc(com.cur);
    val(com.ls[com.cur], v2, dummy);
    with sc.ls[scr].gr do
    begin
      if v1 >= v2 then
      begin
        com_err;
        exit;
      end;
      yfullmode := 0;
      v1lim := v1 * ypower;
      v2lim := v2 * ypower;
    end;
    UpdateVitals;
    Update(scr);
  end;
  end;
end;

procedure do_yoff;
{ Print/change y offset values of waves. }
var
  dummy, i : integer;
  p, r : real;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
  { Print values. }
  begin
    wv_sel(1); { Get wave list. }
    for i := 1 to wv.num do
      with wv.ls[i]^ do
        if sel = 1 then
        begin
          p := power(par.yoffset);
          if p <> 1 then
            s := 'x' + PowerOfTenPrefix(round(log10(p)))
          else
            s := '';
          com_wr_wv(i, 'yoffset ' + makestring(par.yoffset / p, VALMAX,
            VALDEC) + s, COLORMESS);
        end;
    wv_sel_off;
  end
  else if is_num(s) then
  { Change values. }
  begin
    val(s, r, dummy); { Read in value. }
    wv_sel(1);
    sc_sel_off;
    for i := 1 to wv.num do
      with wv.ls[i]^ do
        if sel = 1 then
        begin
          par.yoffset := r;
          sc.ls[screen].sel := 1;
        end;
    wv_sel_off;
    UpdateSel;
  end
  else
    com_err;
end;

procedure do_ysc;
{ Print/change y scale values of waves. }
var
  dummy, i : integer;
  p, r : real;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
  { Print values. }
  begin
    wv_sel(1); { Get wave list. }
    for i := 1 to wv.num do
      with wv.ls[i]^ do
        if sel = 1 then
        begin
          p := power(par.yscale);
          if p <> 1 then
            s := 'x' + PowerOfTenPrefix(round(log10(p)))
          else
            s := '';
          com_wr_wv(i, 'yscale ' + makestring(par.yscale / p, VALMAX,
            VALDEC) + s, COLORMESS);
        end;
    wv_sel_off;
  end
  else if is_num(s) then
  { Change values. }
  begin
    val(s, r, dummy); { Read in value. }
    wv_sel(1);
    sc_sel_off;
    for i := 1 to wv.num do
      with wv.ls[i]^ do
```

```
        if sel = 1 then
        begin
          par.yscale := r;
          sc.ls[screen].sel := 1;
        end;
    wv_sel_off;
    UpdateSel;
  end
  else
    com_err;
end;

end.
```

4.3. fpesai.pas

```
unit FpesAI;
{$M $4000,0,0 }

{ History of modifications (please add to BOTTOM of list!):

  Version 1: Begun 2jun94 BJG.

  Procedures and functions beginning with letters A through I, for the
  program Fpes. Please see main program (fpes.pas) for more detailed
  program modification notes.
}

interface

uses
  FpesVar;

{procedure ACavg;
procedure ACscanInit; }
procedure ADoff(c : integer);
function ADRead(c : integer) : integer;
procedure ADReadAll;
function ADReadAsm(gain, channel : integer) : integer;
function ADReadStrobe : integer;
procedure AreaFitY;
function AreYouSure : boolean;
function AutoGen(w : integer) : integer;
procedure AutoName(w : integer);
function AutoSave(w : integer) : integer;
{procedure BackgroundSubtractionInput; }
function better_div(a, b : integer) : integer;
procedure Blank;
function bs_on : boolean;
procedure BsAdapt;
procedure CalibEnergy(n : integer);
procedure CalibMass1;
procedure CalibMass2;
procedure ChangeCurrentWave(direction : integer);
procedure ChangePar(w : integer);
procedure ChangeParArrow(direction : integer; var par_ptr : par_type_ptr; var
  d, p : integer; w : integer; var changed : boolean);
procedure ChangeStep(var step : real; direction : integer);
procedure Char_to_int(var res : integer; var errcode : integer);
function ChooseColor(w : integer) : word;
procedure ClearArea(x1, y1, x2, y2 : integer);
procedure ClearYAxisStuff(scr : integer);
procedure ClipDot(x, y : real; scr : integer);
procedure ClipLine(x1, y1, x2, y2 : real; scr : integer);
procedure _col;
procedure CreateWave(d, oldw : integer);
function Dataread(name : integer; numchars : integer; w : integer) :
  boolean;
function DerivX(w : integer; p : integer; xaxismode : integer) : real;
procedure disc;
procedure _dly;
procedure DrawAll;
procedure DrawCursor(scr : integer);
procedure DrawCursorInfo;
procedure DrawData(scr : integer);
procedure DrawMessageBox;
procedure DrawOsc;
procedure DrawScreen(scr : integer);
procedure DrawTitle(scr : integer);
procedure DrawVitals;
procedure DrawWaveData;
procedure DrawXAxisStuff(scr : integer);
procedure DrawYAxisStuff(scr : integer);
procedure EraseCursor(scr : integer);
procedure EraseOsc;
procedure EraseWaves;
procedure Error;
procedure ExitProgram;
procedure fft(var data : fft_array_type; nn, isign : integer);
function FileExists(fn : bufstring) : boolean;
function FilenameStart(s : string) : integer;
function FileOpenWrite(var f : text; s : string) : boolean;
function FindWave(scr : integer) : integer;
procedure FitBs;
procedure FitY(d : integer);
procedure fpes_sv(s : bufstring);
function fullpath(s : bufstring) : boolean;
procedure FullView(i : integer);
function GainCode(adgain : integer) : integer;
function get_extension(s : bufstring) : bufstring;
procedure GraphicsMode;
procedure Help;
function info_area(w : integer) : real;
function info_avg(w : integer) : real;
function info_edge(w : integer; p1, p2 : integer; fifty : real) : real;
procedure info_edges(w : integer; vfind : integer);
function info_ht(w : integer) : real;
procedure info_wr(w : integer; r : real; s : bufstring; code : integer);
procedure InitGraphics;
procedure Initialize;
procedure InitializeWave(w : integer);
procedure Init_ts;
procedure Integral;
procedure intelligent_filename(s : bufstring; var aser : bufstring; var
  anum : integer; var ext : bufstring);
procedure IntroduceProgram;
function is_bg(w : integer) : boolean;
function is_num(s : bufstring) : boolean;
function is_sel(s : bufstring; sel : integer) : boolean;

implementation

uses
```

```
  crt, dos, FpesCom, FpesJR, FpesST, FpesUZ, graph, Keys,
  DOSShell, TPDecl;

procedure ADoff(c : integer);
{ Turns off channel c if no other scans using it. }
var
  i : integer;
begin
  { Turn off. }
  ad.ls[c].on := 0;
  { Turn back on if any other wave is using it... }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if scan.mode > 1 then
        case par.dt of
        dt_COR: if par.cor.ch = c then
          ad.ls[c].on := 1;
        dt_POW: if par.pow.ch = c then
          ad.ls[c].on := 1;
        end;
  { ...or if osc is using it. }
  with osc do
    if mode = 1 then
      ad.ls[ch].on := 1;
end;

function ADRead(c : integer) : integer;
{ Reads A/D board channel c. }
const
  ADREADDELAY = 1; { ms to wait after calling ADReadAsm. }
begin
  if dev_rd = 1 then
  begin
    ADRead := ADReadAsm(GainCode(ad.ls[c].gain), c);
    delay(ADREADDELAY);
  end
  else
    ADRead := 0;
end;

procedure ADReadAll;
{ Reads all active channels of A/D board and saves values in array ADresult. }
var
  i : integer; { Channel number. }
begin
  for i := 1 to AD_MAX do
    if ad.ls[i].on = 1 then
      AD.ls[i].result := ADRead(i);
end;

function ADReadAsm(gain, channel : integer) : integer; assembler;
{ Performs A/D conversion on input signal, with external clock signal from
  suitable TTL source (Stanford Box). 16-bit value (only 12 bits valid)
  stored in AX is automatically placed in the function result. Channel
  (0-15) specifies which channel to read. Gain defines the gain of the
  channel: 0 = 1x, 1 = 2x, 2 = 4x, 3 = 8x. }
const
{ DT2821 register addresses. }
  BASE = $240;
  ADCSR = BASE;        { A/D control/status. }
  CHANCSR = BASE + 2; { Channel-gain list control/status. }
  ADDAT = BASE + 4;    { A/D data. }
  DACSR = BASE + 6;    { D/A control/status. }
  DADAT = BASE + 8;    { D/A data. }
  DIODAT = BASE + 10; { DIO data. }
  SUPCSR = BASE + 12; { Supervisory control/status. }
  TMRCTR = BASE + 14; { Pacer clock. }
asm
{         mov    ax, 02e7h     40 kHz
          mov    dx, TMRCTR
          out    dx, ax }
          mov    ax, 2240h
          mov    dx, SUPCSR
          out    dx, ax
          mov    ax, 8000h
          mov    dx, CHANCSR
          out    dx, ax
          mov    ax, gain { Load gain. }
{         mov    bx, 16
          imul   bx { Multiply ax by 16 since gain occupies bits 5&4. }
          add    ax, channel { Add in channel number. }
          sub    ax, 1 { Adjust by -1 since user refers to 1-8, not 0-7.}
          add    ax, 0200h { Add in other flags. }
          mov    dx, ADCSR
          out    dx, ax
          mov    ax, 0000h
          mov    dx, CHANCSR
          out    dx, ax
          mov    ax, 0012h (10h)   { Preload multiplexer (bit 4 = 1),
                                     enable external clock (bit 1 = 1). }
          mov    dx, SUPCSR
          out    dx, ax
          mov    dx, ADCSR    { Wait for multiplexer to settle (bit 8 =
                                0).}
@wait1:   in     ax, dx
          and    ah, 01h
          jnz    @wait1
          mov    ax, 000ah (8h)   { Trigger external clock (bit 3 = 1) to
                                    begin conversion. External clock must be
                                    enabled again (bit 1 = 1). }
          mov    dx, SUPCSR
          out    dx, ax
          mov    dx, ADCSR    { Conversion done when bit 7 = 1. }
@wait2:   in     ax, dx
          and    al, 80h
          jz     @wait2
          mov    dx, ADDAT    { Read data. }
          in     ax, dx
end;

function ADReadStrobe : integer; assembler;
{ Reads signal using internal clock to take a fast scan of signal coming in.
  Currently set at 40 kHz acquisition rate. }
const
{ DT2821 register addresses. }
  BASE = $240;
  ADCSR = BASE;        { A/D control/status. }
  CHANCSR = BASE + 2; { Channel-gain list control/status. }
  ADDAT = BASE + 4;    { A/D data. }
  DACSR = BASE + 6;    { D/A control/status. }
  DADAT = BASE + 8;    { D/A data. }
  DIODAT = BASE + 10; { DIO data. }
  SUPCSR = BASE + 12; { Supervisory control/status. }
  TMRCTR = BASE + 14; { Pacer clock. }
asm
          mov    ax, 02e7h    { 40 kHz }
```

```
            mov     dx, TMRCTR
            out     dx, ax
            mov     ax, 2240h
            mov     dx, SUPCSR
            out     dx, ax
            mov     ax, 8000h
            mov     dx, CHANCSR
            out     dx, ax
            mov     ax, 0230h     { Gain = 8 (bits 5-4 = 11); channel = 1 (bits
                                    3-0 = 0000). }
            mov     dx, ADCSR
            out     dx, ax
            mov     ax, 0000h
            mov     dx, CHANCSR
            out     dx, ax
            mov     ax, 0010h (12h)   { Preload multiplexer (bit 4 = 1), enable ex-
                                        ternal clock (bit 1 = 1). }
            mov     dx, SUPCSR
            out     dx, ax
            mov     dx, ADCSR     { Wait for multiplexer to settle (bit 8 = 0).}
  @wait1:   in      ax, dx
            and     ah, 01h
            jnz     @wait1
            mov     ax, 0008h (ah)    { Trigger external clock (bit 3 = 1) to begin
                                        conversion. External clock must be enabled
                                        again (bit 1 = 1). }
            mov     dx, SUPCSR
            out     dx, ax
            mov     dx, ADCSR     { Conversion done when bit 7 = 1. }
  @wait2:   in      ax, dx
            and     al, 80h
            jz      @wait2
            mov     dx, ADDAT     { Read data. }
            in      ax, dx
end;

procedure AreaFitY;
{ Normalizes all visible waves on current wave's screen by their areas,
  assigning current wave a yscale of 1. }
var
  w,p : integer;
  accum, waccum : real;
begin
  if wv.num = 0 then exit;
  if addwaves.mode = 1 then
  begin
    com_wr('Cannot AreaFit Y in Addwaves Mode.', COLORHL);
    exit;
  end;
  sc.ls[wv.ls[wv.cur]^.screen].gr.yaxismode := YAXISMODE_ABSOLUTE;
  UpdateVLimits;
  UpdateYAxis(wv.ls[wv.cur]^.screen);
  accum := 1;
  with wv.ls[wv.cur]^ do
    for p := 1 to par.pt do
      accum := accum + data[p];
  for w := 1 to wv.num do
    with wv.ls[w]^ do
    if (on = 1) and (screen = wv.ls[wv.cur]^.screen) then
    begin
      waccum := 1;
      for p := 1 to par.pt do
        waccum := waccum + data[p];
      par.yscale := accum / waccum;
      par.yoffset := -vmin * par.yscale;
    end;
{Messagedelay('area=',accum,1); }
  ToggleYaxisMode(wv.ls[wv.cur]^.screen);
end;

function AreYouSure : boolean;
{ Waits for user to answer yes or no, returning value as a boolean. }
begin
  case com_wr_yn('Are you sure') of
    0 : AreYouSure := false;
    1 : AreYouSure := true;
  end;
end;

function AutoGen(w : integer) : integer;
{ Automatically generate new wave based on old if auto.gen = 1, with
  automatic filename if auto.fn = 1 (otherwise assigns '' to fn). If out
  of room, removes old waves if auto.rm = 1. If no new wave is created,
  assigns scan.mode = 0 to prevent overwriting. Returns # of new wave
  created, or 0 if none. }
var
  s : bufstring;
  w2 : integer;
begin
  if auto.gen = 1 then
  begin
    if wv.num = MAXWAVES then
    begin { Remove waves first. }
    end;
    if wv.num < MAXWAVES then { If there's room now. }
    begin
      inc(wv.num);
      wv.ls[wv.num]^ := wv.ls[w]^; { Copy old wave to new. }
      wv.ls[wv.num]^.scan.mode := 0; { Turn off scan mode for old
        wave. }
      wv.ls[w]^.col := ChooseColor(wv.num); { Use color we would have
        given to new wave, since it gets old color. }
      with wv.ls[w]^ do
      begin
        FillChar(data, sizeof(data), 0);
        FillChar(tmp, sizeof(tmp), 0);
        AutoName(w);
      end;
      { Reassign addwaves pointer. }
      if addwaves.w = w then
        addwaves.w := wv.num;
      { Reassign cursor. }
      if wv.cur = w then
      begin
        wv.cur := wv.num;
        ChangeCurrentWave(0);
      end;
      { Reassign background wave pointers. }
  {   for w2 := 1 to wv.num - 1 do
        with wv.ls[w2]^ do
          if par.ele.bg = w then
            par.ele.bg := wv.num; }
      AutoGen := wv.num;
      exit;
    end;
  end
```

```
  else
    AutoGen := 0;
  wv.ls[w]^.scan.mode := 0;
end;

procedure AutoName(w : integer);
{ Assigns automatic filename if auto.fn = 1; otherwise assigns '' as
  filename. }
var
  s : bufstring;
begin
  with wv.ls[w]^ do
    if auto.fn = 1 then
    begin
      str(auto.num, s);
      inc(auto.num);
      par.fn := auto.ser + s + '.' + dt_NAME[par.dt];
    end
    else
      par.fn := '';
end;

function AutoSave(w : integer) : integer;
{ Automatically save wave if auto.sv = 1; otherwise, turn scan.mode off
  for wave. Also calls AutoGen. Returns # of new wave created. }
begin
  if auto.sv = 1 then
  begin
    if SaveWave(w) then
      AutoSave := AutoGen(w);
  end
  else
    wv.ls[w]^.scan.mode := 0;
end;

{procedure BackgroundSubtractionInput;
{ Allows user to change electron scan background subtraction parameters. }
{var
  p : real; { Power. }
{  r : real; { Generic input real. }
{  readonly : boolean; { Flag that scan in progress, so cannot change values. }
{  w : integer; { Wave counter. }
{begin
  readonly := false;
  { Check if any electron scan in progress. }
{  for w := 1 to wv.num do
    if (wv.ls[w]^.par.dt = dt_ELE) and (wv.ls[w]^.scan.mode =
      1) then
      readonly := true;
  textmode;
  clrscr;
  writeln('Background subtraction parameters:');
  with background do
  begin
    p := power(stagedelay);
    r := stagedelay / p;
    writeln('Stage delay (default = ', r : VALMAX : VALDEC, ' ', UnitPrefix(
      round(log10(p))), 's)');
    if not readonly then
    begin
      readln2r(r);
      stagedelay := r * p;
    end;
    p := power(stagepos);
    r := stagepos / p;
    writeln('Stage position for background scan (default = ', r : VALMAX :
      VALDEC, ' ', UnitPrefix(round(log10(p))), 'm)');
    if not readonly then
    begin
      readln2r(r);
      stagepos := r * p;
    end;
  end;
  if readonly then
    waitkey;
  DrawAll;
end; }

function better_div(a, b : integer) : integer;
{ Computes a div b consistently for a > 0 and a < 0. }
begin
  if a >= 0 then
    better_div := a div b
  else
    better_div := (a + 1) div b - 1;
end;

procedure Blank;
{ Blanks data in selected waves using blankmin, blankmax range. Called
  from do_blank. }
var
  i, j : integer;
begin
  { Final check that user wants this. }
  if com_wr_yn('Blank data') = 0 then
    exit;
  sc_sel_off;
  { Blank data, marking screens affected. }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
      begin
        if (scan.mode > 0) or (addwaves.w = i) then
          com_wr_wv(i, 'active: cannot blank!', COLORHL)
        else
        begin
          for j := blankmin to blankmax do
            data[j] := 0;
          sc.ls[screen].sel := 1;
          datasaved := 0;
        end;
      end;
  wv_sel_off;
  UpdateSel;
end;

function bs_on : boolean;
{ Checks waves for any par.ele.bs.mode > 0 and returns true if so. }
var
  i : integer;
begin
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if (par.dt = dt_ELE) and (par.ele.bs.mode > 0) then
      begin
        bs_on := true;
```

```
      exit;
    end;
  bs_on := false;
end;

procedure BsAdapt;
{ Adapts old-style background-subtracted ele scans to new scheme. For
  each bg/fg wave pair, finds total bg counts and places in fg tot,
  adds bg into _bs.bg wave, and adds bg into fg wave. }
var
  i, j : integer;
  tot : longint;
begin
  sc_sel_off;
  with wv.ls[_bs.bg]^ do
  begin
    if (_bs.bg < 1) or (_bs.bg > wv.num) or (par.dt <> dt_ELE) or
      (scan.mode > 0) or ((addwaves.mode = 1) and (addwaves.w =
      _bs.bg)) then
    begin
      com_wr_wv(_bs.bg, 'not valid for bg. Aborted.', COLORHL);
      exit;
    end;
    if (datasaved = 0) or (parsaved = 0) then
      if com_wr_yn('bg wave not saved. Overwrite') = 0 then
      begin
        com_wr('bg adapt aborted.', COLORMESS);
        exit;
      end;
    par.ele.bs.mode := 1; { Set mode. }
    par.ele.bs.tot := 0; { Clear for accumulation to come. }
    fillchar(data, sizeof(data), 0); { Blank wave. }
    fillchar(tmp, sizeof(tmp), 0);
    datasaved := 0; { New data will not have been saved. }
    parsaved := 0;
    sc.ls[screen].sel := 1 { Tag screen for update. }
  end;

  wv_sel(1); { Find participating waves. }

  { Adapt. }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      { Only look at selected fg waves which are not _bs.bg. }
      if (sel = 1) and (par.dt = dt_ELE) and (par.ele.bs.mode > 0) and
        (par.ele.bs.mode <= wv.num) and (i <> _bs.bg) then
      begin
        tot := 0; { Clear accumulator. }
        { Add bg wave to _bs.bg wave. }
        with wv.ls[par.ele.bs.mode]^ do
          for j := 1 to par.pt do
          begin
            wv.ls[_bs.bg]^.data[j] := wv.ls[_bs.bg]^.data[j] +
              data[j]; { Add bg to _bs.bg. }
            wv.ls[i]^.data[j] := wv.ls[i]^.data[j] + data[j]; { Add bg
              to fg. }
            inc(tot, round(data[j])); { Accumulate bg counts. }
          end;
        par.ele.bs.mode := 1; { Can erase bg wave # now. }
        par.ele.bs.tot := tot; { Place total counts in fg tot. }
        wv.ls[_bs.bg]^.par.ele.bs.tot := wv.ls[_bs.bg]^.par.ele.
          bs.tot +  tot; { Add to _bs.bg total. }
        datasaved := 0;
        parsaved := 0;
        sc.ls[screen].sel := 1; { Tag screen for update. }
      end;
  wv_sel_off;
  UpdateSel;
end;

procedure CalibEnergy(n : integer);
{ Accept point n (1 or 2) in a calibration energy spectrum. }
var
  done : boolean;
  p : real; { Power. }
  slope : real; { Intermediate value. }
begin
  if wv.num = 0 then
    exit;
  if (wv.ls[wv.cur]^.par.dt <> dt_ELE) or
    (wv.ls[wv.cur]^.on = 0) or (sc.ls[wv.ls[wv.cur]^.screen].gr.
    cursorvisible = 0) then
    exit;
  textmode;
  clrscr;
  write('Point ', n, '. Use last info ctr result as point reference? (Y/N)');
  done := false;
  repeat
    case readkey of
    'n', 'N':
    begin
      with sc.ls[wv.ls[wv.cur]^.screen].gr do
        if xh.mode = 0 then
          calib.n[n] := PtoU(wv.cur, cursorp, XAXISMODE_NORMAL)
        else
          calib.n[n] := PtoU(wv.cur, UtoP(wv.cur, xh.u[3], 0),
            XAXISMODE_NORMAL);
      done := true;
    end;
    'y', 'Y':
    begin
      calib.n[n] := PtoU(wv.cur, UtoP(wv.cur, info.ctr, 0),
        XAXISMODE_NORMAL);
      done := true;
    end;
    extended : readkey;
    end;
  until done;
  writeln;
  write('Reference time = ');
  p := power(calib.n[n]);
  writevalue(calib.n[n], p, VALDEC, VALMAX, 's');
  writeln;
  write('Enter energy of calibration point (default = ');
  writevalue(calib.c[n], 1, VALDEC, VALMAX, 'eV): ');
  readln2r(calib.c[n]);

  if n = 2 then
  { Calculate calibration parameters, setting quad factor to 0 since none was
    calculated (and, presumably, needed). }
  begin
    with calib do
    begin
      slope := (n[2] - n[1]) / (1 / sqrt(c[2]) - 1 / sqrt(c[1]));
      setenergyconversiont0 := n[1] - slope / sqrt(c[1]);
      setenergyconversionlength := slope / sqrt(SLOPEFACTOR);
```

```
    end;
    { Copy new parameters to current wave and default parameters. }
    pardf[dt_ELE].ele.cal.t0 := setenergyconversiont0;
    pardf[dt_ELE].ele.cal.len := setenergyconversionlength;
    wv.ls[wv.cur]^.par.ele.cal.t0 := setenergyconversiont0;
    wv.ls[wv.cur]^.par.ele.cal.len := setenergyconversionlength;
    writeln('New calibration parameters:');
    p := power(setenergyconversiont0);
    writeln('T0 = ', setenergyconversiont0 / p : VALMAX : VALDEC, ' ',
      UnitPrefix(round(log10(p))), 's');
    p := power(setenergyconversionlength);
    writeln('Length = ', setenergyconversionlength / p : VALMAX : VALDEC, ' ',
      UnitPrefix(round(log10(p))), 'm');
    waitkey;
  end;
  UpdateAll;
end;

procedure CalibMass1;
{ Accept first point in a calibration energy spectrum. }
begin
  if wv.num = 0 then
    exit;
  if (wv.ls[wv.cur]^.par.dt <> dt_MAS) or
    (wv.ls[wv.cur]^.on = 0) or (sc.ls[wv.ls[wv.cur]^.screen].gr.
    cursorvisible = 0) then
    exit;
  textmode;
  clrscr;
  write('Enter mass of first calibration point (Da): ');
  readln(calib.c[1]);
  calib.n[1] := PtoU(wv.cur, sc.ls[wv.ls[wv.cur]^.screen].gr.cursorp,
    XAXISMODE_NORMAL);
  DrawAll;
end;

procedure CalibMass2;
{ Accept second point in a calibration energy spectrum and calculate calibra-
  tion parameters, assigning them to the dt_ELE default. }
var
  p : real; { Power. }
  slope : real; { Intermediate value. }
begin
  if wv.num = 0 then
    exit;
  if (wv.ls[wv.cur]^.par.dt <> dt_MAS) or (wv.ls[wv.cur]^.on
    = 0) or (sc.ls[wv.ls[wv.cur]^.screen].gr.cursorvisible = 0) then
    exit;
  textmode;
  clrscr;
  write('Enter mass of second calibration point (Da): ');
  readln(calib.c[2]);
  calib.n[2] := PtoU(wv.cur, sc.ls[wv.ls[wv.cur]^.screen].gr.cursorp,
    XAXISMODE_NORMAL);
  { Calculate calibration parameters. }
  with calib do
    with wv.ls[wv.cur]^.par.mas.cal do
    begin
      sl := sqr((sqrt(c[2]) - sqrt(c[1])) / (n[2] - n[1]));
      int := n[1] - sqrt(c[1] / sl);
      pardf[dt_MAS].mas.cal.sl := sl;
      pardf[dt_MAS].mas.cal.int := int;
      writeln('New calibration parameters:');
      p := power(int);
      writeln('Intercept = ', int / p : VALMAX : VALDEC, ' ',
        UnitPrefix(round(log10(p))), 's');
      p := power(sl);
      writeln('Slope = ', sl / p : VALMAX : VALDEC, ' ', UnitPrefix(
        round(log10(p))), 'Da/s^2');
    end;
  waitkey;
  UpdateAll;
end;

procedure ChangeCurrentWave(direction : integer);
{ Changes the value of wv.cur; updates the wavedata area of screen. }
var
  temp : integer; { Temp storage for sc.cur. }
begin
  if wv.num = 0 then
    exit;
  if sc.ls[sc.cur].gr.cursorvisible = 1 then
    EraseCursor(sc.cur);
  wv.cur := wv.cur + direction;
  if wv.cur < 1 then
    wv.cur := wv.num
  else if wv.cur > wv.num then
    wv.cur := 1;
  temp := sc.cur;
  sc.cur := wv.ls[wv.cur]^.screen; { Force current screen to be one with
    current wave. }
  DrawTitle(temp);  { Change colors of old and current screen. }
  DrawTitle(sc.cur);
  UpdateCursor(sc.cur);
{  com_wr_scan; { Print scan info if wave is active. }
  DrawWaveData;
  if sc.ls[sc.cur].gr.cursorvisible = 1 then
  begin
    DrawCursorInfo;
    DrawCursor(sc.cur);
  end;
end;

procedure ChangePar(w : integer);
var
  changed: boolean; { Flag indicating if we changed something. If not in
    changedf mode, will update wv.ls[]^.parsaved at end. }
  d : integer; { Current dt. }
  exitflag : boolean; { Flag to exit loop. }
  i : integer; { General integer. }
  p, poffset : integer; { Which parameter is being edited. poffset is
    the parameter offset when list scrolls off-screen. }
  par_ptr : par_type_ptr; { Pointer to parameters (defaults or wave). }
  r : real; { General real for data processing. }
  s : bufstring; { General string for data processing. }
  y : integer; { Y position. }
begin
  if w = 0 then
  begin
    d := dt_MIN;
    par_ptr := @pardf[d];
  end
  else
  begin
    par_ptr := @wv.ls[w]^.par;
    d := par_ptr^.dt;
```

```
  changed := false;
end;
exitflag := false;
p := 1;
poffset := 0;
while not exitflag do
begin
  if sc.mode = sc_mode_GR then
    ShowParams(par_ptr, p, w, poffset); { Draw screen if in graphics
      mode.}
  if keypressed then
  begin
    y := p - poffset - 1 + PARYSTART;
    case readkey of
    CR:
      if (w = 0) or (wv.ls[w]^.scan.mode = 0) then
      with par_ptr^ do
      begin
       if p < USERMIN then
        y := CHANGEPARYSTART + p - 1
      else
        y := CHANGEPARYUSER + p - USERMIN; }
        gotoxy(CHANGEPARXVALUE, y);
        textcolor(LIGHTRED);
        changed := true;
        case p of
        PAR_ALERT:
          begin
            readln2i(alert);
            if alert < 0 then
              alert := 0
            else if alert > 1 then
              alert := 1;
          end;
        PAR_COMMENT:
          begin
            s := '';
            readln(s);
            if s <> '' then
              comment := s;
          end;
        PAR_FILENAME:
          begin
            s := '';
            readln(s);
            if s <> '' then
              fn := s;
          end;
        PAR_GEN:
          begin
            readln2i(gen);
            if gen < 0 then
              gen := 0
            else if gen > 1 then
              gen := 1;
          end;
        PAR_PT:
          begin
            readln2i(pt);
            if pt < 0 then
              pt := 0
            else if pt > MAXPOINTS then
              pt := MAXPOINTS;
          end;
        PAR_PT_GL:
{         if d <> dt_ELE then }
          begin
            readln2i(pt_gl);
            if pt_gl < 0 then
              pt_gl := 0
            else if pt_gl > MAXPOINTS then
              pt_gl := MAXPOINTS;
            case d of
            dt_COR: { Change stage stop automatically. }
              UpdateCORLimits(par_ptr);
            dt_MAS: { Change linked parameters. }
              mas.scantime := timeperpt * pt_gl;
            end;
          end;
        PAR_SCAN:
          begin
            readln2i(scan);
            if scan < 0 then
              scan := 0;
          end;
        PAR_SCAN_GL:
          begin
            readln2i(scan_gl);
            if scan_gl < 0 then
              scan_gl := 0;
          end;
        PAR_SKIP:
          begin
            readln2i(skip);
            if skip < 0 then
              skip := 0;
          end;
        PAR_VSTOP: readvalue(vstop, 0);
        PAR_YOFFSET: readvalue(yoffset, 0);
        PAR_YSCALE:
          begin
            r := yscale; { Save old value of yscale for calc below: }
            readvalue(r, 0);
            { Rescale yoffset to keep same relative position: }
            if (yoffsetrescale = 1) and (r <> 0) then
              yoffset := yoffset * r / yscale;
            yscale := r;
          end
          else
          case d of
          dt_COR:
          case p of
            PAR_C_CHANNEL:
            begin
              readln2i(cor.ch);
              if cor.ch < 1 then
                cor.ch := 1
              else if cor.ch > AD_MAX then
                cor.ch := AD_MAX;
            end;
            PAR_C_SHOTSPERPT:
            begin
              readln2i(cor.shotsperpt);
              if cor.shotsperpt < 1 then
                cor.shotsperpt := 1;
            end;
```

```
    PAR_C_STAGEwobAMPL:
    begin
      readvalue(cor.ts.wob.ampl, POWTS);
      limit(cor.ts.wob.ampl, STAGEMIN, STAGEMAX);
    end;
    PAR_C_STAGEwobPER:
    begin
      readvalue(cor.ts.wob.per, POWTS);
      limit(cor.ts.wob.per, STAGEMIN, STAGEMAX);
    end;
    PAR_C_STAGEwobPH:
    begin
      readvalue(cor.ts.wob.ph, 0);
      limit(cor.ts.wob.ph, TS_PHASE_MIN, TS_PHASE_MAX);
    end;
      PAR_C_STAGESTART:
        begin
          readvalue(cor.ts.start, POWTS);
          roundoff(cor.ts.start, ts.step);
          limit(cor.ts.start, STAGEMIN, STAGEMAX);
          { Update number of points. }
          pt_gl := 1 + integer(round((cor.ts.stop - cor.ts.
            start) / cor.ts.step));
        end;
      PAR_C_STAGESTEP:
       begin
          readvalue(cor.ts.step, POWTS);
          roundoff(cor.ts.step, ts.step);
          limit(cor.ts.step, ts.step, STAGEMAX);
          { Update time per point. }
          timeperpt := cor.ts.step / HALFSPEEDOFLIGHT;
          UpdateCORLimits(par_ptr);
{          pt_gl := 1 + integer(round((cor.ts.stop - cor.ts.
            start) / cor.ts.step)); }
        end;
      PAR_C_STAGESTOP:
        begin
          readvalue(cor.ts.stop, POWTS);
          roundoff(cor.ts.stop, ts.step);
          limit(cor.ts.stop, STAGEMIN, STAGEMAX);
          { Update number of points. }
          pt_gl := 1 + integer(round((cor.ts.stop - cor.ts.
            start) / cor.ts.step));
        end;
      PAR_C_STAGET0:
        begin
          readvalue(cor.ts.t0, POWTS);
          roundoff(cor.ts.t0, ts.step);
          limit(cor.ts.t0, STAGEMIN, STAGEMAX);
        end;
      PAR_TIMEPERPT:
        begin
          readvalue(timeperpt, POWFS);
          { Round to time corresponding to nearest 1 um: }
          roundoff(timeperpt, ts.step / HALFSPEEDOFLIGHT);
          { Limit: }
          limit(timeperpt, ts.step / HALFSPEEDOFLIGHT,
            STAGEMAX / HALFSPEEDOFLIGHT);
          { Change stage step, number of points: }
          cor.ts.step := timeperpt * HALFSPEEDOFLIGHT;
          pt_gl := 1 + integer(round((cor.ts.stop - cor.ts.
            start) / cor.ts.step));
        end;
        end;
    dt_ELE:
      case p of
      PAR_E_BS_LAST: readvalue(ele.bs.last, 0);
      PAR_E_BS_MODE:
      begin
        readln2i(ele.bs.mode);
        if ele.bs.mode < 0 then
          ele.bs.mode := 0
        else if ele.bs.mode > 1 then
          ele.bs.mode := 1;
      end;
      PAR_E_BS_TOT: readvalue(ele.bs.tot, 0);
      PAR_E_CALEV: readvalue(ele.cal.ev, 0);
      PAR_E_CALLENGTH: readvalue(ele.cal.len, 0);
      PAR_E_CALT0: readvalue(ele.cal.t0, 0);
      PAR_E_CALQUAD: readvalue(ele.cal.quad, 0);
      PAR_E_CALQUADOFF: readvalue(ele.cal.quadoff, 0);
      PAR_E_REPRATE : readvalue(ele.reprate, 0);
      PAR_E_SHOTSPERSCAN:
      begin
        readln2i(ele.shotsperscan);
        if ele.shotsperscan < 1 then
          ele.shotsperscan := 1
        else if ele.shotsperscan > MAXSHOTSPERSCAN then
          ele.shotsperscan := MAXSHOTSPERSCAN;
      end;
      PAR_E_STAGEwobAMPL:
      begin
        readvalue(ele.ts.wob.ampl, POWTS);
        limit(ele.ts.wob.ampl, STAGEMIN, STAGEMAX);
      end;
      PAR_E_STAGEwobPER:
      begin
        readvalue(ele.ts.wob.per, POWTS);
        limit(ele.ts.wob.per, STAGEMIN, STAGEMAX);
      end;
      PAR_E_STAGEwobPH:
      begin
        readvalue(ele.ts.wob.ph, 0);
        limit(ele.ts.wob.ph, TS_PHASE_MIN, TS_PHASE_MAX);
      end;
      PAR_E_STAGEPOS:
      begin
        readvalue(ele.ts.pos, POWTS);
        roundoff(ele.ts.pos, ts.step);
        limit(ele.ts.pos, STAGEMIN, STAGEMAX);
        { Update time of FPES. }
        ele.dly := (ele.ts.pos - ele.ts.t0) / HALFSPEEDOFLIGHT;
      end;
      PAR_E_STAGET0:
      begin
        readvalue(ele.ts.t0, POWTS);
        roundoff(ele.ts.t0, ts.step);
        limit(ele.ts.t0, STAGEMIN, STAGEMAX);
        { Update time of FPES. }
        ele.dly := (ele.ts.pos - ele.ts.t0) / HALFSPEEDOFLIGHT;
      end;
      PAR_E_TIMEFPES:
      begin
        readvalue(ele.dly, POWFS);
        { Update stage position. }
        ele.ts.pos := ele.dly * HALFSPEEDOFLIGHT + ele.ts.t0;
```

```
      roundoff(ele.ts.pos, ts.step);
      limit(ele.ts.pos, STAGEMIN, STAGEMAX);
      { Update dly again. }
      ele.dly := (ele.ts.pos - ele.ts.t0) / HALFSPEEDOFLIGHT;
    end;
  end;
  dt_POW:
    case p of
    PAR_P_CALINT: readvalue(pow.cal.int, 0);
    PAR_P_CALSLOPE: readvalue(pow.cal.sl, 0);
    PAR_P_CHANNEL:
    begin
      readln2i(pow.ch);
      if pow.ch < 1 then
        pow.ch := 1
      else if pow.ch > AD_MAX then
        pow.ch := AD_MAX;
    end;
    PAR_TIMEPERPT:
    begin
      readvalue(timeperpt, 0);
      if timeperpt < 0 then
        timeperpt := 0;
    end;
  end;
  dt_MAS:
    case p of
    PAR_M_CALINT: readvalue(mas.cal.int, 0);
    PAR_M_CALSLOPE: readvalue(mas.cal.sl, 0);
    PAR_M_CHANNEL:
    begin
      readln2i(mas.ch);
      if mas.ch < 1 then
        mas.ch := 1
      else if mas.ch > MAS_CH_MAX then
        mas.ch := MAS_CH_MAX;
    end;
    PAR_M_DELAY:
    begin
      readvalue(mas.delay, 0);
      if mas.delay < 0 then
        mas.delay := 0;
    end;
    PAR_M_INV:
    begin
      readln2i(mas.inv);
      if mas.inv < 0 then
        mas.inv := 0
      else if mas.inv > 1 then
        mas.inv := 1;
    end;
    PAR_M_SCANTIME:
    begin
      readvalue(mas.scantime, 0);
      if mas.scantime < 0 then
        mas.scantime := timeperpt;
      { Change linked parameters. }
      pt := round(mas.scantime / timeperpt);
      if pt > MAXPOINTS then
        pt := MAXPOINTS;
      mas.scantime := timeperpt * pt;
    end;
    PAR_M_VERT:
      readvalue(mas.vert, 0);
    PAR_TIMEPERPT:
    begin
      readvalue(timeperpt, 0);
      { Force to valid timeperpt. }
      timeperpt := TEK_TIMEPERPT[TEK_TIMEPERPT_to_code
        (timeperpt)];
      { Change linked parameters. }
      pt := round(mas.scantime / timeperpt);
      if pt > MAXPOINTS then
        pt := MAXPOINTS;
      mas.scantime := timeperpt * pt;
    end;
  end;
end;
end;
ESC: exitflag := true;
EXTENDED:
  case readkey of
    XARROWUP:
      if p > 1 then
      begin
        dec(p);
        if (y = PARYSTART) and (poffset > 0) then
          dec(poffset);
      end;
    XARROWDOWN:
      if p < USERMAXDT[d] then
      begin
        inc(p);
        if (y = PARYSTOP) and (poffset < USERMAXDT[d] - PARYSTOP
          + PARYSTART - 1) then
          inc(poffset);
      end;
    XARROWLEFT: ChangeParArrow(-1, par_ptr, d, p, w, changed);
    XARROWRIGHT: ChangeParArrow(1, par_ptr, d, p, w, changed);
    XEND:
    begin
      p := USERMAXDT[d];
      poffset := p - PARYSTOP + PARYSTART - 1;
    end;
    XHOME:
    begin
      p := 1;
      poffset := 0;
    end;
  end;
end;
ShowParams(par_ptr, p, w, poffset); { Update screen. }
end;
sc.mode := sc_mode_TX_OVR; { Set text "override" flag -- which means
  when Scan is called, if any graphics commands are issued,
  nothing will get though except com_wr using COLORHL as color --
  i.e. important messages. }
Scan; { Keep active waves happy. }
end;
if not (w = 0) and changed then
  wv.ls[w]^.parsaved := 0;
sc.mode := sc_mode_TX; { Allow screen to redraw again. }
UpdateAll;
end;
```

```
procedure ChangeParArrow(direction : integer; var par_ptr : par_type_ptr; var
  d, p : integer; w : integer; var changed : boolean);
{ Increases (direction = 1) or decreases (direction = -1) certain parameters,
  given information about them from ChangePar (par_ptr, d, p). Makes changed
  flag = 1 if valid key is pressed. }
var
  i : integer;
  r : real;
  s : bufstring;
begin
  { Cannot change if scanning. }
  if (w > 0) and (wv.ls[w]^.scan.mode > 0) then
    exit;
  changed := true;
  with par_ptr^ do
  case p of
  PAR_ALERT: alert := 1 - alert;
  PAR_DT:
    begin
      d := d + direction;
      if d < dt_MIN then
        d := dt_MAX
      else if d > dt_MAX then
        d := dt_MIN;
      if w = 0 then
        par_ptr := @pardf[d] { Point to new pardf entry. }
      else
      begin
        s := fn; { Save old filename. }
        par_ptr^ := pardf[d]; { Copy goodies. }
        fn := s; { Restore filename. }
        { If using standard file extension, change to reflect dt. }
        s := get_extension(fn);
        for i := dt_MIN to dt_MAX do
          if s = dt_NAME[i] then
            fn := rm_extension(fn) + '.' + dt_NAME[dt];
      end;
    end;
  PAR_GEN: gen := 1 - gen;
  PAR_SCAN:
    begin
      inc(scan, direction);
      if scan < 0 then
        scan := 0;
    end;
  PAR_SCAN_GL:
    begin
      inc(scan_gl, direction);
      if scan_gl < 1 then
        scan_gl := 1;
    end;
  PAR_SH:
    begin
      inc(sh, direction);
      if (d <> dt_ELE) and (sh = wv_sh_DLY) then
        inc(sh, direction); { Skip this value if not ele. }
      if sh < 1 then
        sh := wv_sh_MAX
      else if sh > wv_sh_MAX then
        sh := 1;
    end;
  PAR_SKIP:
    begin
      inc(skip, direction);
      if skip < 0 then
        skip := 0;
    end;
  else
    case d of
    dt_COR:
      case p of
      PAR_C_CHANNEL:
        begin
          inc(cor.ch, direction);
          if cor.ch < 1 then
            cor.ch := 1
          else if cor.ch > AD_MAX then
            cor.ch := AD_MAX;
        end;
      PAR_C_SHOTSPERPT:
        begin
          inc(cor.shotsperpt, direction);
          if cor.shotsperpt < 1 then
            cor.shotsperpt := 1;
        end;
      PAR_C_STAGESTEP, PAR_TIMEPERPT:
        { Change stage step and time per point at the same time. }
        begin
          r := cor.ts.step;
          if r > 2 * ts.step then
            r := r + direction * ts.step;
          timeperpt := r / HALFSPEEDOFLIGHT;
          cor.ts.step := r;
          { Update number of points. }
          pt_gl := 1 + round((cor.ts.stop - cor.ts.start) / r);
        end;
      end;
    dt_ELE:
      case p of
      PAR_E_BS_MODE:
        ele.bs.mode := 1 - ele.bs.mode;
      PAR_TIMEPERPT:
        begin
          i := TimebaseToCode(timeperpt) + direction;
          if i = -1 + direction then { Number did not match any value in
            array. }
            i := MCS_TIMEPERPT_MIN
          else if i < MCS_TIMEPERPT_MIN then
            i := MCS_TIMEPERPT_MIN
          else if i > MCS_TIMEPERPT_MAX then
            i := MCS_TIMEPERPT_MAX;
          timeperpt := MCS_TIMEPERPT[i];
        end;
      end;
    dt_POW:
      case p of
      PAR_P_CHANNEL:
        begin
          inc(pow.ch, direction);
          if pow.ch < 1 then
            pow.ch := 1
          else if pow.ch > AD_MAX then
            pow.ch := AD_MAX;
        end;
      end;
    dt_MAS:
```

```
      case p of
      PAR_M_INV: mas.inv := 1 - mas.inv;
      PAR_TIMEPERPT:
        begin
          i := TEK_TIMEPERPT_to_code(timeperpt) + direction;
          if i = -1 + direction then { Number did not match any value in
            array. }
            i := TEK_TIMEPERPT_MIN
          else if i < TEK_TIMEPERPT_MIN then
            i := TEK_TIMEPERPT_MIN
          else if i > TEK_TIMEPERPT_MAX then
            i := TEK_TIMEPERPT_MAX;
          timeperpt := TEK_TIMEPERPT[i];
          { Change linked parameters. }
          pt := round(mas.scantime / timeperpt);
          if pt > MAXPOINTS then
            pt := MAXPOINTS;
          mas.scantime := pt * timeperpt;
        end;
      PAR_M_CHANNEL:
        begin
          inc(mas.ch, direction);
          if mas.ch < 1 then
            mas.ch := 1
          else if mas.ch > MAS_CH_MAX then
            mas.ch := MAS_CH_MAX;
        end;
      end;
    end;
  end;
end;

procedure ChangeStep(var step : real; direction : integer);
{ Increases (direction = 1) or decreases (direction = -1) the value of step in
  stages of 1, 2, 5 * (power of 10). }
begin
{ Strip away power of 10 to look only at mantissa: }
  case round(exp(ln(step) - LN_10 * rounddown(ln(step) / LN_10 + SMALL))) of
  1:
    if direction = 1 then
      step := step * 2
    else
      step := step / 2;
  2:
    if direction = 1 then
      step := step * 2.5
    else
      step := step / 2;
  5:
    if direction = 1 then
      step := step * 2
    else
      step := step / 2.5;
  end;
end;

procedure char_to_int(var res : integer; var errcode : integer);
var
  i,j : integer;
  resbuf : Bufstring;

  function is_num(c: char) : boolean;
  begin
    if c in ['0'..'9'] then
      is_num := true
    else is_num := False;
  end;

begin
  i := 1;
  j := 1;
  resbuf := '    ';
  while NOT (is_num(ibbuf[i])) do i := i + 1;
  while (is_num(ibbuf[i])) do
  begin
    resbuf[j] := ibbuf[i];
    j := j+1;
    i := i+1;
  end;
  val(resbuf,res,errcode);
end;

function ChooseColor(w : integer) : word;
{ Looks at wave colors currently in use, and returns a color not already
  used, unless all are taken in which case chooses one from color array
  based on wave number w.}
var
  i : integer;
  sel : array[1 .. COLORMAX] of boolean; { Array to keep track of chosen
    colors. }
begin
  for i := 1 to COLORMAX do { Initialize array. }
    sel[i] := false;
  for i := 1 to wv.num do { Flag all colors used. }
    sel[anticolor[wv.ls[i]^.col]] := true;
  { Look for a free color. }
  for i := 1 to COLORMAX do
    if sel[i] = false then
    begin
      ChooseColor := COLOR[i];
      exit;
    end;
  { No untaken colors; choose based on wave number. }
  ChooseColor := COLOR[(w - 1) mod COLORMAX + 1];
end;

procedure ClearArea(x1, y1, x2, y2 : integer);
{ Clears area of screen specified by coordinates. }
begin
  SetViewPort(x1, y1, x2 - 1, y2 - 1, CLIPOFF);
  ClearViewPort;
  SetViewPort(0, 0, GetMaxX, GetMaxY, CLIPOFF);
end;

procedure ClearYAxisStuff(scr : integer);
{ Clears area containing ylabel, ynumbers and yaxis on screen scr. }
begin
  with sc.ls[scr].gr do
    ClearArea(ylabel.x1, ylabel.y1, yaxis.x2 - 1, ylabel.y2);
end;

procedure ClipDot(x, y : real; scr : integer);
{ Draws dot (on screen scr) only if y coordinate is in plot area. Assumes
  x coordinate already in range. }

begin
```

```
  if (y >= sc.ls[scr].gr.yaxis.y1) and (y <= sc.ls[scr].gr.yaxis.
    y2 - 1) then
  begin
    if ( dotradius = 0 ) then
      PutPixel(round(x), round(y), GetColor)
    else
      FillEllipse(round(x), round(y), dotradius, dotradius);
  end;
end;

procedure ClipLine(x1, y1, x2, y2 : real; scr : integer);
{ Draws segment of line (if any) which falls within y coordinate limits
  of plot area on screen scr. Assumes x coordinates already in range. }
var
  a : real;                          { Generic real. }
  slope : real;                      { Slope of line. }
begin
{ Switch starting and ending coordinates if y1 > y2; this makes case evalua-
  tion much easier. }
  if y1 > y2 then
  begin
    a := x1;
    x1 := x2;
    x2 := a;
    a := y1;
    y1 := y2;
    y2 := a;
  end;
{ Now check to see if line is at all on screen; do nothing otherwise. }
  with sc.ls[scr].gr do
    if (y1 <= yaxis.y2 - 1) and (y2 >= yaxis.y1) then
    begin
    { See if x coordinates are equal; if so, clipping routine is trivial. }
      if x1 = x2 then
      begin
        if y1 < yaxis.y1 then
          y1 := yaxis.y1;
        if y2 > yaxis.y2 - 1 then
          y2 := yaxis.y2 - 1;
      end
      else
      begin
      { See if y1 off screen, and move it on screen if so. }
        if y1 < yaxis.y1 then
        begin
          x1 := x1 + (yaxis.y1 - y1) * (x2 - x1) / (y2 - y1) ;
          y1 := yaxis.y1;
        end;
      {   See if y2 off screen, and move it on screen if so. }
        if y2 > yaxis.y2 - 1 then
        begin
          x2 := x2 + (yaxis.y2 - 1 - y2) * (x2 - x1) / (y2 - y1);
          y2 := yaxis.y2;
        end;
      end;

    { Draw clipped line. }
      Line(round(x1), round(y1), round(x2), round(y2));
    end;
end;

procedure _col;
{ Handle wave color variables. }
var
  c : word; { Color. }
  i : integer;
  s : bufstring;
  w : integer;
begin
  { Just list colors of specified waves. }
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
  begin
    wv_sel(1);
    for w := 1 to wv.num do
      with wv.ls[w]^ do
        if sel = 1 then
          com_wr_wv(w, 'col ' + COLORNAME[col], COLORMESS);
    wv_sel_off;
    exit;
  end
  else if s = 'df' then
    c := 0
  else
  begin
    { Find integer corresponding to color name. }
    c := 0;
    i := 1;
    while (i <= COLORMAX) and (c = 0) do
      if s = COLORNAME[i] then
        c := i
      else
        inc(i);
    { Couldn't match. }
    if c = 0 then
    begin
      com_err;
      exit;
    end;
  end;
  wv_sel(1);

  { Assign colors. }
  sc_sel_off;
  for w := 1 to wv.num do
    with wv.ls[w]^ do
      if sel = 1 then
      begin
        if c = 0 then
          col := COLOR[(w - 1) mod COLORMAX + 1]
        else
          col := c;
        sc.ls[screen].sel := 1;
      end;
  wv_sel_off;

  { Update tagged screens. }
  for i := 1 to sc.num do
    if sc.ls[i].sel = 1 then
      DrawScreen(i);
```

```
  sc_sel_off;
  DrawVitals;
end;

procedure CreateWave(d, oldw : integer);
{ Creates a new wave of datatype d and filename based on wave oldw.
  There are two flag conditions on these variables:
  1. If oldw = 0, the filename determined by AutoName is used (or '' is
     given if auto.fn = 0).
  2. If d = 0, the datatype of wave oldw is used. If oldw = 0 also, the
     default datatype (dt_MIN) is used. }
var
  i : integer;
  s : bufstring;
  w : integer; { Temporary wave number. }
begin
  if wv.num = maxwaves then
  begin
    com_wr('Memory Full.', COLORHL);
    exit;
  end;
  w := wv.num + 1;

  { Generate default wave, or copy contents of oldw, as specified. }
  if (w > 1) and (oldw > 0) and (d = 0) then
  begin
    wv.ls[w]^.par := wv.ls[oldw]^.par;
    { Reset some parameters. }
    wv.ls[w]^.par.yscale := pardf[wv.ls[w]^.par.dt].yscale;
    wv.ls[w]^.par.yoffset := pardf[wv.ls[w]^.par.dt].yoffset;
  end
  else if d > 0 then
    wv.ls[w]^.par := pardf[d]
  else
    wv.ls[w]^.par := pardf[dt_MIN];
  Fillchar(wv.ls[w]^.data, sizeof(wv.ls[w]^.data), 0);
  InitializeWave(w); { Set system defaults for wave. }
  { Change filename. }
  if oldw = 0 then
    AutoName(w)
  else
  begin
    wv.ls[w]^.par.fn := wv.ls[oldw]^.par.fn;
    NextFileNum(w);
  end;
{  UpdateVitals;
  Update(sc.cur); { Update screen (note only one needed since all waves
    are created on sc.cur). }
end;

function dataread(name : integer; numchars : integer; w : integer) :
  boolean;
{ Reads the data from the device 'name' into buffer of wave w. }
var
s : bufstring; { debug }
  ibptr : ^chararray;
begin
  ibptr := @(wv.ls[w]^.tmp);
  ibrd(name, ibptr^, numchars); { Numchars is precisely known, 2 * pt. Data
    is read into the char array ibbuf. Other routines can then read the same
    array as an integer-type array using intbuf^, which is set in Initialize
    to point to ibbuf. }
  if (ibsta and ERR) <> 0 then
  begin
    error;
    dataread := false;
  end
  else
    dataread := true;
end;

function DerivX(w : integer; p : integer; xaxismode : integer) : real;
{ Calculates differential of x for integration routines, etc. This de-
  pends on xaxistype, and dt of wave. }
var
  t : real; { Temporary holder for calculations. }
begin
  with wv.ls[w]^ do
    case xaxismode of
      XAXISMODE_POINTS: DerivX := 1;
      XAXISMODE_NORMAL:
        case par.dt of
          dt_COR: DerivX := par.cor.ts.step;
          dt_ELE, dt_POW, dt_MAS: DerivX := par.timeperpt;
        end;
      XAXISMODE_CONVERT:
        case par.dt of
          dt_COR, dt_POW, dt_MAS: DerivX := par.timeperpt;
          dt_ELE:
          begin
            t := p * par.timeperpt - par.ele.cal.t0;
            DerivX := 2 * SLOPEFACTOR * par.timeperpt *
              sqr(par.ele.cal.len) / (t * t * t);
          end;
        end;
    end;
end;

procedure disc;
{ Handle discriminator command. }
var
  dummy : integer;
  r : real;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
    com_wr('disc ' + makestring(discrim * 1000, VALMAX, VALDEC) +
      ' mV', COLORMESS)
  else
  begin
    val(s, r, dummy);
    r := r / 1000;
    if r < 0 then
      com_err
    else
      discrim := r;
  end;
end;
```

```
procedure _dly;
{ Set time delay of waves. }
var
  dly : real;
  dummy : integer;
  i : integer;
  s : bufstring;
  step : real;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  if s = '?' then
    { Print delays -- done at end of routine. }
    wv_sel(1)
  else
  begin
    step := 0; { Set to 0 unless user specifies. }
    val(s, dly, dummy); { Read delay time. }
    dly := dly * POWFS;
    if com.cur < com.num then
    { Look for optional step. }
    begin
      inc(com.cur);
      s := com.ls[com.cur];
      if s = 'step' then
      begin
        if com.cur = com.num then
        begin
          com_err;
          exit;
        end;
        inc(com.cur);
        val(com.ls[com.cur], step, dummy);
        step := step * POWFS;
      end
      else
        dec(com.cur); { Point back to before wavelist start. }
    end;
    wv_sel(1);

    { Assign delays. }
    for i := 1 to wv.num do
      with wv.ls[i]^ do
        if (sel = 1) and (par.dt = dt_ELE) then
        begin
          { Assign delay by first ensuring ts position will be valid. }
          par.ele.ts.pos := par.ele.ts.t0 + dly * HALFSPEEDOFLIGHT;
          limit(par.ele.ts.pos, STAGEMIN, STAGEMAX);
          par.ele.dly := (par.ele.ts.pos - par.ele.ts.t0) /
            HALFSPEEDOFLIGHT;
          { Increment delay. }
          dly := dly + step;
        end;
  end;
  { Now print delays. }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
        begin
          if par.dt <> DT_ELE then
            com_wr_wv(i, 'not ELE!', COLORHL)
          else
            com_wr_wv(i, 'dly ' + makestringint(par.ele.dly / POWFS) +
              ' fs', COLORMESS);
        end;
  wv_sel_off;
  DrawWaveData; { Update display. }
end;

procedure DrawAll;
{ Sets up complete screen display for program. }
var
  scr : integer; { Screen counter. }
begin
  if sc.mode = sc_mode_TX_OVR then
    exit;
  GraphicsMode;
  DrawVitals;
  for scr := 1 to sc.num do
    DrawScreen(scr);
end;

procedure DrawCursor(scr : integer);
{ Draws cursor on screen scr - must call after plotting data and calling
  UpdateCursor! }
begin
  if sc.cur <> scr then
    exit;
  if sc.mode <> sc_mode_GR then
    exit;
  with sc.ls[scr].gr do
  begin
    if xh.mode = 0 then
    begin
      if cursorvisible = 1 then
      begin
        { Save area underneath in bitmap: }
        GetImage(cursorx, cursory1, cursorx, cursory2, bitmap^);
        { Draw cursor: }
        SetColor(WHITE);
        line(cursorx, cursory1, cursorx, cursory2);
      end;
    end
    else
      with xh do
      begin
        { Save areas underneath crosshairs: }
        GetImage(x[1], plotarea.y1, x[1], plotarea.y2, bitmap.x[1]^);
        GetImage(x[2], plotarea.y1, x[2], plotarea.y2, bitmap.x[2]^);
        GetImage(x[3], plotarea.y1, x[3], plotarea.y2, bitmap.x[3]^);
        GetImage(plotarea.x1, y[1], plotarea.x2, y[1], bitmap.y[1]^);
        GetImage(plotarea.x1, y[2], plotarea.x2, y[2], bitmap.y[2]^);
        GetImage(plotarea.x1, y[3], plotarea.x2, y[3], bitmap.y[3]^);
        SetColor(WHITE);
        { Draw active crosshairs: }
        SetLineStyle(SolidLn, SolidLn, NormWidth);
        line(x[which], plotarea.y1, x[which], plotarea.y2);
        line(plotarea.x1, y[which], plotarea.x2, y[which]);
        { Draw inactive crosshairs: }
        SetLineStyle(DottedLn, DottedLn, NormWidth);
        line(x[3 - which], plotarea.y1, x[3 - which], plotarea.y2);
```

```
        line(plotarea.x1, y[3 - which], plotarea.x2, y[3 - which]);
        { Draw center crosshairs: }
        SetLineStyle(CenterLn, CenterLn, NormWidth);
        line(x[3], plotarea.y1, x[3], plotarea.y2);
        line(plotarea.x1, y[3], plotarea.x2, y[3]);
        { Reset line style to normal before exiting: }
        SetLineStyle(SolidLn, SolidLn, NormWidth);
      end;
  end;
end;

procedure DrawCursorInfo;
{ All number lengths are VALMAX. Two formats:
Cursor mode (xh.mode = 0):

012345678901234567890123456789
Point  Normal     Converted
XXXX  -XXX.XX us -XXX.XX kDa ( units are examples only )
Intensity     Relative Int.
XXX.XXx10^XX -XXX.XXx10^XX

Crosshairs mode (xh.mode = 1):

012345678901234567890123456789
     X:Converted Y:Relative
Dif. XXX.XX meV  XXX.XXx10^XX
Ctr.-XXX.XX meV -XXX.XXx10^XX
}
var
  p : real;                          { Power of 1000 of r. }
  r : real;                          { Generic real. }
  s : bufstring;                     { Generic string. }
begin
  if wv.num = 0 then
    exit;
  if sc.mode <> sc_mode_GR then
    exit;
  with sc.ls[sc.cur].gr do
  begin
    SetTextJustify(LEFTTEXT, TOPTEXT);
    SetTextStyle(DEFAULTFONT, HORIZDIR, 1);
    SetColor(WHITE);
    with cursorinfo do
    begin
      ClearArea(x1, y1, x2, y2);
      if xh.mode = 0 then
      begin
        if (cursorvisible = 1) and (wv.ls[wv.cur]^.screen = sc.cur) then
          { second check should not be necessary, but doesn't hurt }
        begin
          { MC information. }
          if mc.s <> '' then
            if length(mc.s) < cursorinfo.xmax then
              outtextxy(x1, y1, mc.s)
            else
              outtextxy(x1, y1, copy(mc.s, 1, cursorinfo.xmax - 2) +
                '..');
          { X information. }
          outtextxy(x1, y1 + textsize, 'Point  Normal     Converted');
          { Point. }
          str(cursorp, s);
          outtextxy(x1, y1 + 2 * textsize, s);
          { Normal. }
          r := PtoU(wv.cur, cursorp, XAXISMODE_NORMAL);
          p := Power(r);
          if r < 0 then
            s := makestring(r / p, 7, 5)
          else
            s := ' ' + makestring(r / p, 6, 5);
          outtextxy(x1 + 6 * textsize, y1 + 2 * textsize, s + ' ' +
UnitPrefix(round(
            log10(p))) + XUNITTYPE[wv.ls[wv.cur]^.par.dt, XAXISMODE_NORMAL]
            );
          { Converted. }
          r := PtoU(wv.cur, cursorp, XAXISMODE_CONVERT);
          { Patch to prevent 0 fs from showing up as 10^-24 s. }
          if (xaxisdt = dt_COR) and (abs(r) < POWFS) then
            r := 0;
          p := Power(r);
          if r < 0 then
            s := makestring(r / p, 7, 5)
          else
            s := ' ' + makestring(r / p, 6, 5);
          outtextxy(x1 + 17 * textsize, y1 + 2 * textsize, s + ' ' +
UnitPrefix(round(
            log10(p))) + XUNITTYPE[wv.ls[wv.cur]^.par.dt, XAXISMODE_CONVERT]
            );

          { Y information. }
          outtextxy(x1, y1 + 3 * textsize, 'Intensity     Relative Int.');
          { Intensity. }
          r := PtoV(wv.cur, cursorp, xaxismode, YAXISMODE_ABSOLUTE);
          p := Power(r);
          if r < 0 then
            s := makestring(r / p, 7, 5)
          else
            s := ' ' + makestring(r / p, 6, 5);
          if p <> 1 then
            s := s + 'x' + PowerOfTenPrefix(round(log10(p)));
          outtextxy(x1, y1 + 4 * textsize, s);
          { Relative intensity. }
          r := PtoV(wv.cur, cursorp, xaxismode, YAXISMODE_RELATIVE);
          p := Power(r);
          if r < 0 then
            s := makestring(r / p, 7, 5)
          else
            s := ' ' + makestring(r / p, 6, 5);
          if p <> 1 then
            s := s + 'x' + PowerOfTenPrefix(round(log10(p)));
          outtextxy(x1 + 13 * textsize, y1 + 4 * textsize, s);
        end;
      end
      else

        { xh mode }
        with xh do
        begin
          case xaxismode of
          XAXISMODE_POINTS: s := 'Points';
          XAXISMODE_NORMAL: s := 'Normal';
          XAXISMODE_CONVERT: s := 'Converted';
          end;
          OutTextXY(x1 + 5 * textsize, y1, 'X:' + s);
          case yaxismode of
          YAXISMODE_ABSOLUTE: s := 'Absolute';
          YAXISMODE_RELATIVE: s := 'Relative';
```

```
        end;
        OutTextXY(x1 + 17 * textsize, y1, 'Y:' + s);
        { X difference. }
        r := abs(u[2] - u[1]);
        p := Power(r);
        s := makestring(r / p, VALMAX - 1, VALDEC);
        if xaxismode = XAXISMODE_POINTS then
        begin
          if p <> 1 then
            s := s + 'x' + PowerOfTenPrefix(round(log10(p)));
        end
        else
          s := s + ' ' + UnitPrefix(round(log10(p))) + XUNITTYPE[
            xaxisdt, xaxismode];
        OutTextXY(x1, y1 + textsize, 'Dif. ' + s);
        { Y difference. }
        r := abs(v[2] - v[1]);
        p := Power(r);
        s := makestring(r / p, VALMAX - 1, VALDEC);
        if p <> 1 then
          s := s + 'x' + PowerOfTenPrefix(round(log10(p)));
        OutTextXY(x1 + 17 * textsize, y1 + textsize, s);
        { X center. }
        r := u[3];
        p := Power(r);
        s := makestring(r / p, VALMAX, VALDEC);
        if xaxismode = XAXISMODE_POINTS then
        begin
          if p <> 1 then
            s := s + 'x' + PowerOfTenPrefix(round(log10(p)));
        end
        else
          s := s + ' ' + UnitPrefix(round(log10(p))) + XUNITTYPE[
            xaxisdt, xaxismode];
        OutTextXY(x1, y1 + 2 * textsize, 'Ctr.' + s);
        { Y center. }
        r := v[3];
        p := Power(r);
        if r < 0 then
          s := makestring(r / p, VALMAX, VALDEC)
        else
          s := ' ' + makestring(r / p, VALMAX - 1, VALDEC);
        if p <> 1 then
          s := s + 'x' + PowerOfTenPrefix(round(log10(p)));
        OutTextXY(x1 + 16 * textsize, y1 + 2 * textsize, s);
      end;
    end;
  end;
end;

procedure DrawData(scr : integer);
{ Draw wave data on screen scr. }
var
  p : integer;
  p1 : integer;
  p2 : integer;
  temp : FillPatternType;
  w : integer;
  x1 : real;
  x2 : real;
  y1 : real;
  y2 : real;
begin
  if sc.mode <> sc_mode_GR then
    exit;
  with sc.ls[scr].gr do
  begin
    for w := 1 to wv.num do
      if (wv.ls[w]^.on = 1) and (wv.ls[w]^.screen = scr) then
      begin
        SetColor(wv.ls[w]^.col);
        GetFillPattern(temp);
        SetFillPattern(temp, GetColor);
        { Find point limits based on screen limits. }
        p1 := UtoP(w, ullim, 1); { Round up to nearest point in unit space. }
        p2 := UtoP(w, u2lim, -1); { Round down to nearest point in unit space. }

        { Must pass two tests: if p1 = 0, then entire wave is to left of screen
          boundaries; we can forget it: }
          if p1 > 0 then
          begin
            x1 := PtoX(w, p1);
            y1 := PtoY(w, p1);
          { Otherwise, if entire wave is to right of screen boundaries, OR view
            is zoomed in so far that the first point larger than ullim is off
            the right edge of screen, then we shouldn't plot either. We check
            this by looking at x1: }
            if x1 < xaxis.x2 - 1 then
            begin
            { In business! }
              if wv.ls[w]^.lines = 1 then { Draw lines. }
                if (xaxisdt = dt_ELE) and (xaxismode =
                  XAXISMODE_CONVERT) then { Go backward for energy. }
                  for p := p1 downto p2 do
                  begin
                    x2 := PtoX(w, p);
                    y2 := PtoY(w, p);
                    ClipLine(x1, y1, x2, y2, scr);
                    x1 := x2;
                    y1 := y2;
                  end
                else
                  for p := p1 to p2 do
                  begin
                    x2 := PtoX(w, p);
                    y2 := PtoY(w, p);
                    ClipLine(x1, y1, x2, y2, scr);
                    x1 := x2;
                    y1 := y2;
                  end
              else { Plot dots. }
                if (xaxisdt = dt_ELE) and (xaxismode =
                  XAXISMODE_CONVERT) then { Go backward for energy. }
                  for p := p1 downto p2 do
                    ClipDot(PtoX(w, p), PtoY(w, p), scr)
                else
                  for p := p1 to p2 do
                    ClipDot(PtoX(w, p), PtoY(w, p), scr);
            end;
          end;
        end;
    end;
  end;
end;

procedure DrawMessageBox;
```

```
var
  s : bufstring;
begin
  if sc.mode <> sc_mode_GR then
    exit;
  tx_dr(@com.tx);
end;

procedure DrawOsc;
{ Draw virtual oscilloscope (if on). }
begin
  if sc.mode <> sc_mode_GR then
    exit;
  with osc do
  begin
    if mode = 0 then
      exit;
    y := round(VtoY(AD.ls[ch].result, scr)); { Compute y value. }
    with sc.ls[scr].gr.plotarea do
    begin
      GetImage(x1, y, x2, y, bit^); { Save area underneath. }
      SetColor(WHITE);
      ClipLine(x1, y, x2, y, scr); { Draw line. }
    end;
  end;
end;

procedure DrawScreen(scr : integer);
{ Draw axes and belonging waves for screen scr. }
begin
  if sc.mode <> sc_mode_GR then
    exit;
  with sc.ls[scr].gr do
    ClearArea(bdy[1].x, bdy[1].y, bdy[2].x, bdy[2].y);
  DrawData(scr);
  DrawCursor(scr);
  DrawTitle(scr);
  DrawXAxisStuff(scr);
  DrawYAxisStuff(scr);
end;

procedure DrawTitle(scr : integer);
{ Draw title if enabled on screen scr. }
var
  temp : bufstring;
begin
  if sc.mode <> sc_mode_GR then
    exit;
  { Draw screen number in LL corner. }
  with sc.ls[scr].gr do
  begin
    if (xon = 1) or (yon = 1) then
      ClearArea(ylabel.x1, xlabel.y2 - textsize, ylabel.x1 + textsize,
        xlabel.y2);
    { Make current screen title highlighted. }
    if scr = sc.cur then
      SetColor(COLORHL)
    else
      SetColor(WHITE);
    SetTextJustify(LEFTTEXT, TOPTEXT);
    str(scr, temp);
    if (xon = 1) or (yon = 1) then
      Outtextxy(ylabel.x1, xlabel.y2 - textsize, temp);
  end;
  { Draw title if on -- also highlighted if current screen. }
  with sc.ls[scr].ti do
    if on = 1 then
    begin
      SetTextJustify(CENTERTEXT, TOPTEXT);
{       if scr = sc.cur then
        SetColor(COLORHL)
      else
        SetColor(WHITE); }
{      str(scr, temp); }
      ClearArea(bdy[1].x, bdy[1].y, bdy[2].x, bdy[2].y);
      outtextxy((bdy[1].x + bdy[2].x) div 2, bdy[1].y, temp + ': ' + s);
    end;
  SetColor(WHITE); { Reset color to default for next operation. }
  SetTextJustify(LEFTTEXT, TOPTEXT);
end;

procedure DrawVitals;
{ Draw all information except screens. }
begin
  DrawMessageBox;
  DrawWaveData;
  DrawCursorInfo;
end;

procedure DrawWaveData;
{ Print wave information. Indicator variables for each wave are as follows:

  -X.XX *Nis/MDS filename

  where:
  -X.XX is par.yscale (only displayed in relative y axis mode);
  * indicates current wave;
  N is wave number;
  i indicates invisible;
  s indicates wave has been saved;
  / indicates tagged for selection;
  M is mode: if addwave, 'A'; if scanning, 'S';
  S is screen number;
  filename is self-explanatory.
}
var
  s : bufstring; { General string. }
  w : integer; { Wave counter. }
  x, y : integer; { Screen coordinate temp stg. }
begin
  if sc.mode <> sc_mode_GR then
    exit;
  SetTextJustify(LEFTTEXT, TOPTEXT); { Make x,y point to upper left corner of
    string. }
  SetTextStyle(DEFAULTFONT, HORIZDIR, 1); { Select font, direction, and
    size. }
  with wavedata do
  begin
    ClearArea(x1, y1, x2, y2);
    SetColor(WHITE);
    s := VERSION + '  ';
    case _bs.dis of
      0 :  s := s + '    ';
      1 :  s := s + ' DIS';
    end;
    case bs_on of
```

```
      false : s := s + '    ';
      true : s := s + ' BS';
    end;
    case _bs.sts of
      0 : s := s + '    ';
      1 : s := s + ' STS';
    end;
    OutTextXY(x1, y1, s);
  end;
  for w := 1 to wv.num do
  begin
    s := WaveInfo(w, wavedata.xmax div 2);
    SetColor(wv.ls[w]^.col);
    if w <= MAXWAVES div 2 then { First column. }
    begin
      x := wavedata.x1;
      y := wavedata.y1 + w * textsize;
    end
    else
    begin
      x := wavedata.x1 + textsize * wavedata.xmax div 2;
      y := wavedata.y1 + (w - MAXWAVES div 2) * textsize;
    end;
    OutTextXY(x, y, s);
    { Draw box around active scan wave. }
    if (w = scanwave) and (wv.ls[w]^.scan.mode > 0) then
    begin
      line(x, y - 1, x + textsize * wavedata.xmax div 2 - 1, y - 1);
      line(x, y + textsize - 1, x + textsize * wavedata.xmax div 2 - 1,
        y + textsize - 1);
      line(x, y, x, y + textsize - 2);
      line(x + textsize * wavedata.xmax div 2 - 1, y, x + textsize *
        wavedata.xmax div 2 - 1, y + textsize - 2);
    end;
  end;
end;

procedure DrawXAxisStuff(scr : integer);
{ Handles all graphics for x axis on screen scr. }
var
  i : integer;                          { Generic integer. }
  s : bufstring;                        { Generic string. }
  x : integer;                          { Generic screen x coordinate. }
  u : real;                             { Generic user-space x coordinate. }
begin
  if sc.mode <> sc_mode_GR then
    exit;
  with sc.ls[scr].gr do
  begin
    SetTextJustify(CENTERTEXT, TOPTEXT);
    SetTextStyle(DEFAULTFONT, HORIZDIR, 1); { Select font, direction, and
      size. }
    SetColor(WHITE);

  { Draw axis. }
    with plotarea do
    begin { Note line coords are 1 pixel outside active drawing area in all
        directions. }
      line(x1 - 1, y1 - 1, x2, y1 - 1); { Top line. }
      line(x1 - 1, y2, x2, y2); { Bottom line. }
    end;

    if xon = 1 then
    begin
  { Draw tick marks and numbers. }
    u := u1num; { X position in user-space of first tick mark and number. }
    repeat
      with xaxis do
      begin
        x := round(UtoX(u, scr)); { Calculate screen x coordinate. }
        line(x, y1, x, y2 - 2); { Draw tick mark. -2 is to ensure there is
          a space between end of tick mark and beginning of text underneath. }
      end;
      OutTextXY(x, xnumbers.y1, makestring(u / xpower, maxxdigits,
        xdecimals)); { makestring() prevents string from exceeding length
        MAXXDIGITS. }
      u := u + ustep;
    until u > u2num + ustep * SMALL; { Due to small accumulation of error with
      each add, I allow a generous margin of error at end, though this is still
      only 0.1% larger than exact end point. }

    { Draw x label. }
    with xlabel do
      OutTextXY((x1 + x2) div 2, y1, xlabelstring + xunits);
    end;
  end;
end;

procedure DrawYAxisStuff(scr : integer);
{ Handles all graphics for y axis on screen scr. }
var
  s : bufstring;                        { Generic string. }
  y : integer;                          { Generic screen y coordinate. }
  v : real;                             { Generic user-space y coordinate. }
begin
  if sc.mode <> sc_mode_GR then
    exit;
  with sc.ls[scr].gr do
  begin
    SetTextJustify(RIGHTTEXT, CENTERTEXT);
    SetTextStyle(DEFAULTFONT, HORIZDIR, 1); { Select font, direction, and
      size. }
    SetColor(WHITE);

  { Draw axis. }
    with plotarea do
    begin { Note line coords are 1 pixel outside active drawing area in all
        directions. }
      line(x1 - 1, y1 - 1, x1 - 1, y2); { Left line. }
      line(x2, y1 - 1, x2, y2); { Right line. }
    end;

    if yon = 1 then
    begin
  { Draw tick marks and numbers. }
    v := v1num; { Y position in user-space of first tick mark and number. }
    repeat
      with yaxis do
      begin
        y := round(VtoY(v, scr));
        line(x1 + 1, y, x2 - 1, y); { Draw tick mark. +1 is to ensure there is
          a space between end of tick mark and beginning of text to the left. }
      end;
      OutTextXY(ynumbers.x2 - 1, y, makestring(v / ypower, MAXYDIGITS,
        ydecimals)); { makestring() prevents string from exceeding length
        MAXXDIGITS. }
```

```
      v := v + vstep;
    until v > v2num + vstep * SMALL; { Due to small accumulation of error with
      each add, I allow a generous margin of error at end, though this is still
      only 0.1% larger than exact end point. }


  { Draw y label. Note that there seems to be a bug with vertical printing:
    neither LEFTTEXT nor RIGHTTEXT works properly: LEFTTEXT prints 8 pixels to
    the left of expected; RIGHTTEXT prints 1 pixel to the left - i.e., they
    both print at the same position, given an x value. }
    SetTextJustify(LEFTTEXT, CENTERTEXT);
    SetTextStyle(DEFAULTFONT, VERTDIR, 1);
    with ylabel do
      OutTextXY(x2, (y1 + y2) div 2, ylabelstring + yunits);
    end;

    { Reset text parameters for next operation. }
    SetTextJustify(LEFTTEXT, TOPTEXT);
    SetTextStyle(DEFAULTFONT, HORIZDIR, 1);
  end;
end;

procedure EraseCursor(scr : integer);
{ Erase cursor on screen scr. Must already have checked for cursorvisible
  = 1, wv.num > 0. }
begin
  if sc.cur <> scr then
    exit;
  if sc.mode <> sc_mode_GR then
    exit;
{ Restore bitmap(s) recorded in DrawCursor. }
  with sc.ls[scr].gr do
  begin
    if xh.mode = 0 then
    begin
      if cursorvisible = 1 then
        PutImage(cursorx, cursory1, bitmap^, NORMALPUT)
    end
    else with xh do
    begin
      PutImage(x[1], plotarea.y1, bitmap.x[1]^, NORMALPUT);
      PutImage(x[2], plotarea.y1, bitmap.x[2]^, NORMALPUT);
      PutImage(x[3], plotarea.y1, bitmap.x[3]^, NORMALPUT);
      PutImage(plotarea.x1, y[1], bitmap.y[1]^, NORMALPUT);
      PutImage(plotarea.x1, y[2], bitmap.y[2]^, NORMALPUT);
      PutImage(plotarea.x1, y[3], bitmap.y[3]^, NORMALPUT);
    end;
  end;
end;

procedure EraseOsc;
{ Erase virtual oscilloscope (if on). }
begin
  if sc.mode <> sc_mode_GR then
    exit;
  with osc do
  begin
    if mode = 0 then
      exit;
    PutImage(sc.ls[scr].gr.plotarea.x1, y, bit^, NORMALPUT);
  end;
end;
```

```
procedure EraseWaves;
{ Erase all tagged waves. For each erasure, must move waves above it
  down, if any. Keeps cursor the same unless it was on deleted wave, in
  which case, it inherits wave moved into its place, or previous wave if
  none. Marks affected screen. }
var
  active : boolean; { Flag indicating wave is active. }
  s : bufstring;
  w, w2 : integer; { Wave counters. }
begin
  for w := wv.num downto 1 do { Must count backward to work. }
    if wv.ls[w]^.sel = 1 then
    begin
      str(w, s);
      { Check to be sure wave is not active. }
      active := false;
      if wv.ls[w]^.scan.mode > 0 then
        active := true
      else if (addwaves.mode = 1) and (w = addwaves.w) then
        active := true;
      if active then
        com_wr('Cannot remove active wave ' + s + '!', COLORHL)
      else
      begin
        { Keep various wave pointers on correct wave: }
        if wv.cur >= w then
          dec(wv.cur);
        if addwaves.w > w then { Cannot delete addwave. }
          dec(addwaves.w);
        if scanwave > w then { Cannot delete scanning wave. }
          dec(scanwave);
        if mon.w = w then
          mon.w := 0 { Turn off mon. }
        else if mon.w > w then
          dec(mon.w);
        if _bs.bg >= w then
          dec(_bs.bg); { Ok if this goes to 0. }
        if _bs.fg >= w then
          dec(_bs.fg); { Ok if this goes to 0. }
{        for w2 := 1 to wv.num do
          with wv.ls[w2]^.par do
            if dt = dt_ELE then
              if ele.bg > w then
                dec(ele.bg);     }
        sc.ls[wv.ls[w]^.screen].sel := 1; { Tag screen. }
        for w2 := w to wv.num - 1 do { Perform erase. }
          wv.ls[w2]^ := wv.ls[w2 + 1]^;
        dec(wv.num);
      end;
      wv.ls[w]^.sel := 0; { Turn off wave tag. }
    end;
  { Patch to prevent cursor from vanishing. }
  if (wv.num > 0) and (wv.cur = 0) then
    wv.cur := 1;
end;

procedure error;
{ Prints error message to screen fault is found during ibrd. }
begin
  if (ibsta and ERR) <> 0 then
    com_wr('Found an error. Redoing scan.', COLORHL);
{   begin
      Textmode;
```

```
      Writeln(#7,'Found an error.');
      writeln('ibsta = ', ibsta);
      Writeln('iberr = ', iberr);
      if iberr = EDVR then writeln (' EDVR <DOS Error>');
      if iberr = ECIC then writeln (' ECIC <Not CIC>');
      if iberr = ENOL then writeln (' ENOL <No Listener>');
      if iberr = EADR then writeln (' EADR <Address error>');
      if iberr = EARG then writeln (' EARG <Invalid argument>');
      if iberr = ESAC then writeln (' ESAC <Not Sys Ctrl>');
      if iberr = EABO then writeln (' EABO <Op. Aborted>');
      if iberr = ENEB then writeln (' ENEB <No GPIB boards>');
      if iberr = EOIP then writeln (' EOIP <Async I/O in prg>');
      if iberr = ECAP then writeln (' ECAP <No Capability>');
      if iberr = EFSO then writeln (' EFSO <File Sys. Error>');
      if iberr = EBUS then writeln (' EBUS <Command Error>');
      if iberr = ESTB then writeln (' ESTB <Status Byte Lost>');
      if iberr = ESRQ then writeln (' ESRQ <SRQ stuck on>');
      if iberr = ETAB then writeln (' ETAB <Table Overflow>');
      writeln('ibcnt = ', ibcnt);
      delay(2000);
      Graphicsmode;
      DrawAll;
    end; }
end;

procedure ExitProgram;
{ Check if user wants to exit, then checks if waves are saved, allowing
  user one more chance to exit if there are unsaved waves. }
var
  i : integer;
begin
  if com_wr_yn('Exit program') = 0 then
    exit;
  exitflag := true;
  { Check that all waves are saved. Search until we find one which
    isn't. }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if (datasaved = 0) or (parsaved = 0) then
      begin
        if com_wr_yn('Not all waves saved. Still exit') = 0 then
          exitflag := false;
        exit;
      end;
end;

procedure fft(var data : fft_array_type; nn, isign : integer);
{ Fast Fourier transform procedure of data. nn is number of points.
  isign indicates direction of Fourier transform, 1 or -1. }
var
  ii, jj, n, mmax, m, j, istep, i : integer;
  wtemp, wr, wpr, wpi, wi, theta : real;
  tempr, tempi : real;
begin
  n := 2 * nn;
  j := 1;
  for ii := 1 to nn do
  begin
    i := 2 * ii - 1;
    if (j > i) then begin
      tempr := data[j];
      tempi := data[j + 1];
      data[j] := data[i];
      data[j + 1] := data[i + 1];
      data[i] := tempr;
      data[i + 1] := tempi;
    end;
    m := n div 2;
    while (m >= 2) and (j > m) do begin
      j := j - m;
      m := m div 2;
    end;
    j := j + m;
  end;
  mmax := 2;
  while n > mmax do begin
    istep := 2 * mmax;
    theta := TWOPI * isign / mmax;
    wpr := -2.0 * sqr(sin(0.5 * theta));
    wpi := sin(theta);
    wr := 1.0;
    wi := 0.0;
    for ii := 1 to (mmax div 2) do begin
      m := 2 * ii - 1;
      for jj := 0 to (n - m) div istep do begin
        i := m + jj * istep;
        j := i + mmax;
        tempr := wr * data[j] - wi * data[j + 1];
        tempi := wr * data[j + 1] + wi * data[j];
        data[j] := data[i] - tempr;
        data[j + 1] := data[i + 1] - tempi;
        data[i] := data[i] + tempr;
        data[i + 1] := data[i + 1] + tempi;
      end;
      wtemp := wr;
      wr := wr * wpr - wi * wpi + wr;
      wr := wi * wpr + wtemp * wpi + wi;
    end;
    mmax := istep;
  end;
end;

function FileExists(fn : bufstring) : boolean;
{ Checks whether or not file 'fn' exists, returning TRUE if so, FALSE if
  not. }
var
  f : file; { File variable. }
begin
  assign(f, fn);
  {$I-}
  reset(f);
  {$I+}
  if IOResult = 0 then
  begin
    FileExists := TRUE;
    close(f);
  end
  else
    FileExists := FALSE; { File never opened if IOResult <> 0. }
end;

{function find_extension(s : string) : integer;
{ Returns numeric extension, or 0 if non-numeric. }
{var
  i : integer;
  v : integer;
```

```
  d : integer; { Dummy code. }
(begin
  i := length(s) - 1;
  while (i > 1) and (i > length(s) - 4) and (s[i] <> '.') do
    i := i - 1;
  v := 0;
  if s[i] = '.' then
    val(copy(s, i + 1, length(s) - i), v, d);
  find_extension := v;
end; }

function FilenameStart(s : string) : integer;
{ Finds position in s where filename part starts; all text to left of this
  point is the path. }
var
  i : integer;
begin
  for i := length(s) downto 1 do
    case s[i] of
    '\', ':' :
      begin
        FilenameStart := i + 1;
        exit;
      end;
    end;
  FilenameStart := 1;
end;

function FileOpenWrite(var f : text; s : string) : boolean;
{ Opens file for writing using error suppression to detect disk errors
  without crashing program. }
begin
  assign(f, s);
  {$I-}
  rewrite(f);
  {$I+}
  if IOResult = 0 then
    FileOpenWrite := true
  else
  begin
    com_wr('Cannot open ' + s, COLORHL);
    FileOpenWrite := false;
  end;
end;

function FindWave(scr : integer) : integer;
{ Returns number of first visible wave on screen scr. Otherwise returns
  0. }
var
  w : integer;
begin
  { Return 0 if no waves. }
  if wv.num = 0 then
  begin
    FindWave := 0;
    exit;
  end;
  { If current wave on screen, use this in preference to first wv.ls. }
  if wv.ls[wv.cur]^.screen = scr then
  begin
    FindWave := wv.cur;
    exit;
  end;
  { Look for first wave on screen scr which is visible. }
  for w := 1 to wv.num do
    with wv.ls[w]^ do
      if (screen = scr) and (on = 1) then
      begin
        FindWave := w;
        exit;
      end;
  { Give up: return 0. }
  FindWave := 0;
end;

procedure FitBs;
{ Scales all bs ele scans by 1/tot. }
var
  i : integer;
  r : real;
begin
  { Find wave to scale to: use current if valid bs ele; otherwise use
    first one. }
  r := 1; { Set as default to avoid /0 error. }
  with wv.ls[wv.cur]^ do
    if (par.dt = dt_ELE) and (par.ele.bs.mode = 1) then
      r := par.ele.bs.tot
    else
      for i := 1 to wv.num do
        with wv.ls[i]^ do
          if (par.dt = dt_ELE) and (par.ele.bs.mode = 1) then
          begin
            r := par.ele.bs.tot;
            { Avoid /0 error: }
            if r < 1 then
              r := 1
            else
              i := wv.num; { Trick to get out of loop. }
          end;

  { Scale other waves. }

  sc_sel_off; { Clear screen sel tags. }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if (par.dt = dt_ELE) and (par.ele.bs.mode = 1) then
      begin
        par.yoffset := 0;
        if par.ele.bs.tot > 0 then
          par.yscale := r / par.ele.bs.tot
        else
          par.yscale := 0; { This will signal user that tot = 0. }
        sc.ls[screen].sel := 1; { Tag screen. }
        sc.ls[screen].gr.yaxismode := YAXISMODE_RELATIVE;
      end;
  UpdateSel;
end;

procedure FitY(d : integer);
{ Changes wv.ls[]^.par.yoffset and .yscale of each visible wave,
  so limits all fit on screen together (like fullview, but in
  relative y mode). Applies to screen containing wave w only. }
{ Using current wave, or first qualifying wave if current wave doesn't
  qualify, as a reference, changes yoffset and yscale of other visible
  waves of datatype d (or all datatypes if d < dt_MIN) so that they all
  have same v limits. }
```

```
var
  i : integer; { General counter. }
  wref : integer; { Wave reference #. }
  vrange, vrangecur : real; { Temporary vmax - vmin. }
begin
  if wv.num = 0 then
    exit;
  if addwaves.mode = 1 then
  begin
    com_wr('Cannot fit Y in addwaves mode.', COLORHL);
    exit;
  end;
  { Find all qualifying waves, changing their screens to absolute mode,
    marking screens for later, and recording first qualifying wave with
    wref. }
  sc_sel_off;
  wref := 0;
  for i := wv.num downto 1 do { Go backward so wref = first wave. }
    with wv.ls[i]^ do
    if (on = 1) and ((d < dt_MIN) or (par.dt = d)) then
    begin
      wref := i;
      sel := 1;
      sc.ls[screen].sel := 1;
      sc.ls[screen].gr.yaxismode := YAXISMODE_ABSOLUTE;
    end;
  { Switch reference to wv.cur if it qualifies. }
  if wv.ls[wv.cur]^.sel = 1 then
    wref := wv.cur;
  { Exit if no waves qualify. }
  if wref = 0 then
    exit;
  { Recalculate wave v limits. }
  UpdateVLimits;
  { Calculate reference limits. }
  with wv.ls[wref]^ do
    vrangecur := vmax - vmin;
    if vrangecur < 1 then
      vrangecur := 1; { Prevent yscale from being 0; I use stipulation
        that vrangecur < 1 to prevent problem of roundoff error making
        it not exactly 0, but still close enough to make yscale tiny. }
  { Scale all other waves to fit this range. }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
      begin
        vrange := vmax - vmin;
        if vrange < 1 then
          vrange := 1; { Same idea as above, but now to prevent / 0. }
        par.yscale := vrangecur / vrange;
        par.yoffset := -vmin * par.yscale;
      end;
  { Now go back to relative mode for all screens and update: }
  wv_sel_off;
  for i := 1 to sc.num do
    with sc.ls[i] do
      if sel = 1 then
        gr.yaxismode := YAXISMODE_RELATIVE;
  UpdateSel;

  { Make current screen one with current wave (call ChangeCurrentWave in
    order to update some other variables, too). }
{   ChangeCurrentWave(0);
  { Work with real y values. }
{   sc.ls[sc.cur].gr.yaxismode := YAXISMODE_ABSOLUTE;
  UpdateVLimits;
  UpdateYAxis(sc.cur);
  { Find wave with largest y range. }
{   with wv.ls[wv.cur]^ do
    vrangecur := vmax - vmin;
    if vrangecur < 1 then
      vrangecur := 1; { Prevent yscale from being 0; I use stipulation
        that vrangecur < 1 to prevent problem of roundoff error making
        it not exactly 0, but still close enough to make yscale tiny. }
  { Scale all other waves to fit this range. }
{   for w := 1 to wv.num do
    with wv.ls[w]^ do
      if on = 1 then
      begin
        vrange := vmax - vmin;
        if vrange < 1 then
          vrange := 1; { Same idea as above, but now to prevent / 0. }
{         par.yscale := vrangecur / vrange;
        par.yoffset := -vmin * par.yscale;
      end;
  ToggleYAxisMode(sc.cur); }
end;

procedure fpes_sv(s : bufstring);
{ Save parameter file for fpes macro to file s. }
var
  bg : integer;
  f : text;
  i : integer;
begin
  { Go thru active files and ensure that if _bs.bg > 0, and there are
    intervening waves which are not bs mode, or not ELE datatype, that
    _bs.bg is decreased appropriately. }
  UpdateBg; { Make sure bg is consistent with current waves. }
  bg := _bs.bg; { Assign to temp variable. }
  wv_sel_off;
  if bg > 0 then
  begin
    { Now go through waves up to bg and decrease value of bg for
      every one which is not a bs ELE wave. Also, highlight waves
      whose delays will be written to file. }
    for i := 1 to bg - 1 do
      with wv.ls[i]^ do
        if (par.dt = dt_ELE) and (par.ele.bs.mode > 0) then
          sel := 1
        else
          dec(bg);
    { Continue past bg wave. }
    for i := _bs.bg to wv.num do
      with wv.ls[i]^ do
        if (par.dt = dt_ELE) and (par.ele.bs.mode > 0) then
          sel := 1;
  end
  else { Highlight waves for non-bs file. }
    for i := 1 to wv.num do
      with wv.ls[i]^ do
        if par.dt = dt_ELE then
          sel := 1;

  { Prepare filename, ask for permission to overwrite if it exists. }
  if get_extension(s) = '' then
```

```
    s := s + '.' + FPES_EXT;
{  if not fullpath(s) then
    s := dir + s; } { Use home directory instead of data directory. }
  if fileexists(s) then
    if com_wr_yn('File ' + s + ' exists. Overwrite') = 0 then
      exit;

  { Write file. }
  assign(f, s);
  rewrite(f);
  writeln(f, bg);
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
      begin
        writeln(f, par.ele.dly / POWFS);
        com_wr('wv ' + makestringint(i) + ' dly saved.', COLORMESS);
      end;
  close(f);
  wv_sel_off;
  DrawWaveData;
end;

function fullpath(s : bufstring) : boolean;
{ Determines if filename s contains a full path or is just an extension of the
  current path. Criteria:

  Return TRUE if s[1] = '\' (root reference) or s[2] = ':' (drive reference).
  Return FALSE otherwise.
}
begin
  if (s[1] = '\') or ((length(s) > 1) and (s[2] = ':')) then
    fullpath := true
  else
    fullpath := false;
end;

procedure FullView(i : integer);
{ Quick command to make both x and y axes full-width on screen i. }
begin
  sc.ls[i].gr.xfullmode := 1;
  sc.ls[i].gr.yfullmode := 1;
  Update(i);
end;

function GainCode(adgain : integer) : integer;
{ Calculates gain code used by ADReadAsm:
  adgain  GainCode
  1       0
  2       16
  4       32
  8       48
}
begin
  case adgain of
    2: GainCode := 16;
    4: GainCode := 32;
    8: GainCode := 48;
    else GainCode := 0; { Unity gain for 1, plus any garbage codes. }
  end;
end;

function get_extension(s : bufstring) : bufstring;
{ Returns file extension of filename s. }
var
  i : integer;
begin
  i := length(s);
  while (s[i] <> '.') and (i >= length(s) - 3) do
    dec(i);
  { Check to ensure file extension exists! }
  if i >= length(s) - 3 then
    get_extension := copy(s, i + 1, length(s) - i)
  else
    get_extension := '';
end;

procedure GraphicsMode;
{ Restore screen to graphics mode after being initialized (see
  InitGraphics). }
begin
  SetGraphMode(graphmode);
  sc.mode := sc_mode_GR;
end;

procedure Help;
{ Displays help screen from file HELP_FN. }
var
  e : boolean; { Exit flag. }
  f : text;
  i : integer; { General counter. }
  l : integer; { Line counter. }
  lines : integer; { Max. lines in help file. }
  s : bufstring;
begin
  { Check to make sure file exists. }
  if FileExists(HELP_FN) = false then
  begin
    com_wr('Help file not available.', COLORHL);
    exit;
  end;

  { Determine number of lines in help file. }
  assign(f, HELP_FN);
  reset(f);
  lines := 0;
  repeat
    readln(f, s);
    inc(lines);
  until eof(f);
  close(f);

  { Initialize other variables. }
  TextMode;
  e := false;
  l := 1;

  repeat
    clrscr;
    TextColor(WHITE);
    assign(f, HELP_FN);
    reset(f);
    { Skip lines before current section. }
    for i := 1 to l - 1 do
      readln(f, s);
    for i := 1 to HELP_LINESPERPAGE do
```

```
      if not eof(f) then
      begin
        readln(f, s);
        writeln(s);
      end;
    close(f);
    textcolor(LIGHTMAGENTA);
    write('Use up/down, page up/down, home/end to scroll. Press ESC to exit.');

    { Now wait for user input. }
    case readkey of
    ESC: e := true;
    EXTENDED:
      case readkey of
      XARROWDOWN:
        if l < lines - HELP_LINESPERPAGE + 1 then
          inc(l);
      XARROWUP:
        if l > 1 then
          dec(l);
      XEND: l := lines - HELP_LINESPERPAGE + 1;
      XHOME: l := 1;
      XPAGEDOWN:
        begin
          inc(l, HELP_LINESPERPAGE - 1);
          if l > lines - HELP_LINESPERPAGE + 1 then
            l := lines - HELP_LINESPERPAGE + 1;
        end;
      XPAGEUP:
        begin
          dec(l, HELP_LINESPERPAGE - 1);
          if l < 1 then
            l := 1;
        end;
      end;
  until e;
  DrawAll;
end;

function info_area(w : integer) : real;
{ Calculates the area of wave w. Uses variable binning depending on
  xaxismode and dt. }
var
  i : integer;
  p1, p2 : integer;
begin
  info.area := 0;
  with wv.ls[w]^ do
    with sc.ls[screen] do
    begin
      if (screen = sc.cur) and (gr.xh.mode = 1) then
      { Full range. }
      begin
        p1 := 1;
        p2 := par.pt;
      end
      else
        { Use crosshairs to specify range. }
        if gr.xh.u[1] < gr.xh.u[2] then
        begin
          p1 := UtoP(w, gr.xh.u[1], 1);
          p2 := UtoP(w, gr.xh.u[2], -1);
        end
        else
        begin
          p1 := UtoP(w, gr.xh.u[2], -1);
          p2 := UtoP(w, gr.xh.u[1], 1);
        end;
      for i := p1 to p2 do
        info.area := info.area + PtoV(w, i, gr.xaxismode, gr.yaxismode)
          * DerivX(w, i, gr.xaxismode);
    end;
  info_area := info.area;
end;

function info_avg(w : integer) : real;
{ Calculates the average of wave w. Requires area to be known (call
  info_area). }
var
  dx : real; { x-axis difference. }
begin
  info_avg := 0;
  with wv.ls[w]^ do
  if par.pt > 0 then
  begin
    dx := abs(PtoU(w, par.pt, sc.ls[screen].gr.xaxismode) - PtoU(w,
      pmin, sc.ls[screen].gr.xaxismode)); { Use abs to take into account
      the reversal in orientation for energy space. }
    info_area(w);
    if dx > 0 then
      info_avg := info.area / dx;
  end;
end;

function info_edge(w : integer; p1, p2 : integer; fifty : real) : real;
{ Calculates position of edge, given point limits p1, p2, and 50%
  intensity value fifty. }
var
  p : integer;
  x1, x2, y1, y2 : real;

  procedure loop;
  begin
    with sc.ls[wv.ls[w]^.screen].gr do
    begin
      x2 := PtoU(w, p, xaxismode);
      y2 := PtoV(w, p, xaxismode, yaxismode);
      if (y1 <= fifty) and (y2 >= fifty) then
      begin
        if y1 <> y2 then
          info_edge := (x1 * (y2 - fifty) + x2 * (fifty - y1)) /
            (y2 - y1)
        else
          info_edge := (x1 + x2) / 2;
        p := p2; { Trick to get out out loop. }
      end
      else
      begin
        x1 := x2;
        y1 := y2;
      end;
    end;
  end;

begin
```

```
  with sc.ls[wv.ls[w]^.screen].gr do
  begin
    x1 := PtoU(w, p1, xaxismode);
    y1 := PtoV(w, p1, xaxismode, yaxismode);
    if p1 < p2 then
      for p := p1 + 1 to p2 do
        loop
    else
      for p := p1 - 1 downto p2 do
        loop;
  end;
end;

procedure info_edges(w : integer; vfind : integer);
{ Calculates the left and right 50% edge positions of wave w, and also
  the center and fwhm (average and difference of edge positions, stor-
  ing all in the info record variable. }
var
  fifty : real;
  p1, p2, ptemp : integer;
  rtemp : real;
  v1, v2 : real;
begin
  with info do
  begin
    fwhm := 0;
    ctr := 0;
    edgel := 0;
    edger := 0;
    with wv.ls[w]^ do
    begin
      if (sc.num <> screen) or (sc.ls[screen].gr.xh.mode = 0) then
      { Consider whole wave. }
      begin
        p1 := pmin;
        p2 := par.pt;
        fifty := (vmin + vmax) / 2;
      end
      else
      { Only focus on part of wave inside crosshairs. }
      with sc.ls[screen].gr do
      begin
        if xh.u[1] < xh.u[2] then
        begin
          p1 := UtoP(w, xh.u[1], 1);
          p2 := UtoP(w, xh.u[2], -1);
        end
        else
        begin
          p1 := UtoP(w, xh.u[2], -1);
          p2 := UtoP(w, xh.u[1], 1);
        end;
        if vfind = 0 then
          fifty := xh.v[3] { Center of crosshairs. }
        else
        { Find vlimits within crosshairs. }
        begin
          if p1 > p2 then
          { Switch so 'for' loop will work. }
          begin
            ptemp := p1;
            p1 := p2;
            p2 := ptemp;
          end;
          v1 := PtoV(w, p1, xaxismode, yaxismode);
          v2 := v1;
          for ptemp := p1 + 1 to p2 do
          begin
            rtemp := PtoV(w, ptemp, xaxismode, yaxismode);
            if rtemp < v1 then
              v1 := rtemp;
            if rtemp > v2 then
              v2 := rtemp;
          end;
          fifty := (v1 + v2) / 2;
        end;
      end;
      if (p1 > 0) and (p1 <= par.pt) and (p2 > 0) and (p2 <= par.pt)
        then
      begin
        if p1 > p2 then
        { Switch so 'for' loop will work. }
        begin
          ptemp := p1;
          p1 := p2;
          p2 := ptemp;
        end;
        edgel := info_edge(w, p1, p2, fifty);
        edger := info_edge(w, p2, p1, fifty);
        if edgel > edger then
        { Switch. }
        begin
          rtemp := edgel;
          edgel := edger;
          edger := rtemp;
        end;
        fwhm := abs(edger - edgel);
        ctr := (edger + edgel) / 2;
      end;
    end;
  end;
end;

function info_ht(w : integer) : real;
{ Calculates the height (difference between max and min) of wave w. }
begin
  with wv.ls[w]^ do
    info_ht := vmax - vmin;
end;

procedure info_wr(w : integer; r : real; s : bufstring; code :
  integer);
{ Write quantity r with label s. w is wave number. Code tells whether or
  not to write units after quantity. }
var
  p : real; { Power. }
  s2 : bufstring;
begin
  p := Power(r);
  with wv.ls[w]^ do
  begin
    s2 := XUNITTYPE[par.dt, sc.ls[screen].gr.xaxismode];
    if (s2 = '') or (code = 0) then
    begin
      if p <> 1 then
        s2 := 'x' + PowerOfTenPrefix(round(log10(p)))
```

```
      else
        s2 := '';
    end
    else
      s2 := ' ' + UnitPrefix(round(log10(p))) + s2;
  end;
  com_wr_wv(w, s + ' ' + makestring(r / p, VALMAX, VALDEC) + s2,
    COLORMESS);
end;

procedure InitGraphics;
{ Launch graphics mode. This procedure is copied from Turbo Pascal Reference
  Manual, pp. 138-139 }
var
  errorcode : integer;
  graphdriver : integer;
  maxcolor : integer; { debugging only! }
begin
  graphdriver := DETECT;
  InitGraph(graphdriver, graphmode, 'D:\BP\BGI');
  errorcode := GraphResult;
  if errorcode <> GROK then
  begin
    writeln('Graphics error :', GraphErrorMsg(errorcode));
    writeln('Program aborted...');
    halt(1);
  end;
  textsize := TextHeight('x'); { Number of pixels for a single character,
    used to lay out screen correctly irrespective of graphics mode used. The
    function call requires a dummy string to work. }
  SetBkColor(BLACK); { Set background color to black (all modes). }
  { Set boxes for different information on screen. A pictorial outline is as
    follows (not to scale!):

                +----------++-----------------+
                |cursorinfo||                 |
                |          ||    wavedata     |
                +----------+|                 |
                |messagebox||                 |
                ++--++-----++---------------+-+
                ||  ++                      | |
ylabel---------->|  ||                      | |
                ||  ||       plot area      | |
ynumbers---------> ||                       | |
                ||  ||                      | |
yaxis------------->|                       | |
                ||  ||                      | |
                ||  ++----------------------+ |
                +++-++--------------------^--+-+
screen #--------> +-------------^--------|----+
                +-+----^--------|--------|----+
                       |        |        |
                    xlabel  xnumbers  xaxis
  }
  with cursorinfo do
  begin
    xmax := 29;
    ymax := 5;
    x1 := 0;
    x2 := xmax * textsize; { Number of characters needed. }
    y1 := 0;
    y2 := ymax * textsize;
  end;
  with wavedata do
  begin
    x1 := cursorinfo.x2 + textsize; { Leave one space for readability. }
    x2 := GetMaxX + 1;
    xmax := (x2 + 1 - x1) div textsize;
    ymax := MAXWAVES div 2 + 1;
    y1 := cursorinfo.y1;
    y2 := cursorinfo.y1 + ymax * textsize;
    maxwavenamelength := xmax div 2 - 14; { Define number of characters
      that can be displayed for wave names. }
  end;
  with messagebox do
  begin
    x1 := cursorinfo.x1;
    x2 := cursorinfo.x2;
    y1 := cursorinfo.y2;
    y2 := wavedata.y2;
    xmax := (x2 + 1 - x1) div textsize;
    ymax := (y2 + 1 - y1) div textsize;
  end;
  { New: add this information to the com block. }
  com.tx.bdy[1].x := messagebox.x1;
  com.tx.bdy[1].y := messagebox.y1;
  com.tx.bdy[2].x := messagebox.x2;
  com.tx.bdy[2].y := messagebox.y2;
  com.tx.num.x := messagebox.xmax;
  com.tx.num.y := messagebox.ymax;
  { Calculate maximum room for all screens and store in sc.bdy: }
  sc.bdy[1].x := 0;
  sc.bdy[2].x := GetMaxX + 1;
  sc.bdy[1].y := wavedata.y2;
  sc.bdy[2].y := GetMaxY + 1;
end;

procedure Initialize;
{ Gets program running. }
var
  i : integer;
begin
  { Set up graphics housekeeping. Note that I seem to get error messages
    if I've already printed to text screen before I call InitGraphics. }
  InitGraphics;

  { Reserve memory for variables. }
  GetMem(bitmap, imagesize(0, 0, 0, CURSORLENGTH)); { Cursor bitmap
    buffer. }
  for i := 1 to sc_MAX do { Graphics screen crosshair bitmap buffers. }
    with sc.ls[i].gr do
    begin
      GetMem(xh.bitmap.x[1], imagesize(0, sc.bdy[1].y, 0, sc.
        bdy[2].y));
      GetMem(xh.bitmap.x[2], imagesize(0, sc.bdy[1].y, 0, sc.
        bdy[2].y));
      GetMem(xh.bitmap.x[3], imagesize(0, sc.bdy[1].y, 0, sc.
        bdy[2].y));
      GetMem(xh.bitmap.y[1], imagesize(sc.bdy[1].x, 0, sc.
        bdy[2].x, 0));
      GetMem(xh.bitmap.y[2], imagesize(sc.bdy[1].x, 0, sc.
        bdy[2].x, 0));
      GetMem(xh.bitmap.y[3], imagesize(sc.bdy[1].x, 0, sc.
        bdy[2].x, 0));
    end;
  for i := 1 to MAXWAVES do { Waves. }
```

```
    New(wv.ls[i]);
  GetMem(com.tx.buf, com.tx.num.x * com.tx.num.y); { Com text buffer. }
  GetMem(com.tx.col, 2 * com.tx.num.y); { Com color buffer. }
  GetMem(osc.bit, imagesize(sc.bdy[1].x, 0, sc.bdy[2].x, 0)); { Virtual
    oscilloscope bitmap buffer. }

  { Declaration odds & ends. }
  for i := 1 to COLORMAX do { Color-to-code array. }
    anticolor[COLOR[i]] := i;
  exitflag := false;
  newworkspace := 1;
  ts.pos := 0;
  macro_override := 0;

  { Set up com variables: blank out buffers, put cursor in upper left. }
  FillChar(com.tx.buf^, com.tx.num.x * com.tx.num.y, ' ');
  for i := 1 to com.tx.num.y do
    com.tx.col^[i] := COLORUSER;
  com.tx.cur.x := 1;
  com.tx.cur.y := 1;

  { Set default command to null (in case user initiates session with !! }
  com.num := 0;
  com.sv := 'x';
  com.old := '';

  { See if user wants to disable devices; otherwise open some. }
  textmode;
  clrscr;
  TextColor(WHITE);
  writeln('Disable devices for analysis use only (Yes/No)?');
  dev_rd := -1;
  repeat
    case readkey of
      'y', 'Y': dev_rd := 0;
      'n', 'N': dev_rd := 1;
      EXTENDED: readkey;
    end;
  until dev_rd <> -1;

  { Open channel to translation stage. }
  if dev_rd = 1 then
  begin
    assign(com2, 'COM2');
    rewrite(com2);
  end;

  { Read workspace. }
  ws_rd(WS_FN_DF);
end;

procedure InitializeWave(w : integer);
{ Set wave system parameters to their defaults when a wave is first created or
  read in. }
begin
  with wv.ls[w]^ do
  begin
    col := ChooseColor(w);
    datasaved := 1;
    lines := 1;
    massl := 0;
    on := 1;
    parsaved := 1;
    pmin := 1;
    savemode := 0;
    screen := sc.cur; { Place on current screen. }
    scan.mode := 0;
    sel := 0;
    fillchar(tmp, sizeof(tmp), 0);
  end;
  { Other things which should be done for newly loaded/created wave. }
  wv.num := w; { Now safe to increase wave number. }
  wv.cur := w; { Make cursor point to new wave. }
  ChangeCurrentWave(0); { Update screen #. }
  sc.ls[sc.cur].gr.cursorvisible := 1; { Force cursor to be vi-
    sible for NewCursor. }
  NewCursor; { Place cursor in middle of wave. }
  sc.ls[wv.ls[wv.cur]^.screen].gr.xaxisdt :=
    wv.ls[wv.cur]^.par.dt; { Force most recently loaded
    wave to be the screen's current dt; this is only
    important if xaxismode <> POINTS. }
end;

procedure Init_ts;
{ Initialize translation stage when load new workspace. }
begin
  MakeLookup; { Recalculate lookup table. }
  if dev_rd = 1 then
  begin
    { Set current acceleration and velocity. }
    write(COM2, '#FCAB6', LF, 'B', makestringint(ts.acc), LF);
    delay(ACKDELAY);
    write(COM2, ACK);
    delay(ACKDELAY);
    write(COM2, '#FCAB7', LF, 'B', makestringint(ts.vel), LF);
  end;
end;

procedure Integral;
{ Crosshairs mode only: Calculates integral (sum, actually) of region bounded
  by x-axis xh, printing result in message box. Currently not set up
  to properly integrate in energy-space, so gives error message. }
var
  accum : real; { Integral. }
  p : real; { Power. }
  pt, p1, p2 : integer; { Point counters. }
  s : bufstring; { String. }
  u, u1, u2 : real; { U-space counter. }
begin
  if wv.num = 0 then
    exit;
  with sc.ls[sc.cur].gr do
  begin
    if xh.mode = 0 then
    begin
      com_wr('Must be in xh mode.', COLORHL);
      exit;
    end;
    if wv.ls[wv.cur]^.screen <> sc.cur then
    begin
      com_wr('Current wave must be on screen.', COLORHL);
      exit;
    end;
    if (xaxisdt = dt_ELE) and (xaxismode = XAXISMODE_CONVERT)
      then
    begin
```

```
      com_wr('Cannot integrate in energy space.', COLORHL);
      exit;
    end;
    { Copy xh limits to own variables. }
    u1 := xh.u[1];
    u2 := xh.u[2];
    { Ensure u1 <= u2. }
    if u1 > u2 then
    begin
      u := u1;
      u1 := u2;
      u2 := u;
    end;
    p1 := UtoP(wv.cur, u1, 1);
    p2 := UtoP(wv.cur, u2, -1);
    accum := 0;
    for pt := p1 to p2 do
      accum := accum + PtoV(wv.cur, pt, xaxismode, yaxismode);
    p := Power(accum);
    str(accum / p : VALMAX : VALDEC, s);
    com_wr('Integral = ' + s + 'x' + PowerOfTenPrefix(round(Log10(p))),
      COLORMESS);
  end;
end;

procedure intelligent_filename(s : bufstring; var aser : bufstring; var
  anum : integer; var ext : bufstring);
{ Analyzes string s for series, number and extension parts. Returns
  series in aser, number in anum and extension in ext. }
var
  dash : boolean;
  dummy : integer;
  i : integer;
begin
  { Get EXT if it exists. }
  ext := get_extension(s);
  if ext <> '' then
    s := copy(s, 1, length(s) - length(ext) - 1); { Omit extension from
      string. }
  { See if we have NUM only, or SER-NUM. }
  dash := false;
  for i := length(s) downto 2 { don't check for '-' in first character:
    could be a directory symbol } do
    if s[i] = '-' then
    begin
      { Extract auto.ser and auto.num. }
      aser := copy(s, 1, i);
      val(copy(s, i + 1, length(s)), anum, dummy);
      i := 2; { Trick to get out of loop. }
      dash := true;
    end;
  { Just get number; use auto.ser for series prefix. }
  if dash = false then
  begin
    aser := auto.ser;
    val(copy(s, 1, length(s)), anum, dummy);
  end;
end;

procedure IntroduceProgram;
{ Print friendly greeting. }
const
  CEN = 40;
begin
  textmode;
  clrscr;
  TextColor(WHITE);
  writeln;
  writeln;
  writeln_ctr(HEAD_WV, CEN);
  writeln_ctr(HEAD_WS, CEN);
  writeln;
  writeln_ctr('Jeff Greenblatt', CEN);
  writeln_ctr('Gordon Burton', CEN);
  writeln_ctr('Martin Zanni', CEN);
  writeln;
  writeln_ctr('Data Acquisition and File Comparison Program.', CEN);
  writeln_ctr('Supports file versions E and later.', CEN);
  writeln;
  writeln;
  writeln;
  writeln;
  writeln;
  writeln_ctr('Press F2 to get to system control menu, or any other key to start.',
    CEN);
  if readkey = EXTENDED then
    if readkey = XF2 then
      SystemControl;
end;

function is_bg(w : integer) : boolean;
{ Checks if any wave is calling w its background wave. }
var
  i : integer;
begin
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if (par.dt = dt_ELE) and (par.ele.bs.mode = w) then
      begin
        is_bg := true;
        exit;
      end;
  is_bg := false;
end;

function is_num(s : bufstring) : boolean;
{ Determines if s is a number. }
var
  i : integer;
begin
  for i := 1 to length(s) do
    if ((s[i] < '0') or (s[i] > '9')) and (s[i] <> '+') and (s[i] <>
      '-') and (s[i] <> '.') and (s[i] <> 'e') and (s[i] <> 'E') then
    begin
      is_num := false;
      exit;
    end;
  is_num := true;
end;

function is_sel(s : bufstring; sel : integer) : boolean;
{ Determines if s is a valid sel option (number, 'all' or 'sel'). If
  sel = 0, 'sel' keyword is invalid (can't use 'sel' in SCLIST). }
begin
  if is_num(s) or (s = 'all') or ((sel = 1) and (s = 'sel')) then
    is_sel := true
```

```
  else
    is_sel := false;
end;

end.
```

4.4. fpesjr.pas

```
unit FpesJR;
{ History of modifications (please add to BOTTOM of list!):

  Version 1: Begun 2jun94 BJG.

  Procedures and functions beginning with N through Z for the program FPES.
  See FPES (at appropriate version) for more specific program modification
  notes.
}

interface

uses
  FpesVar;

procedure limit(var r : real; min, max : real);
function Log10(r : real) : real;
procedure MakeFilter(res : real);
procedure MakeLookup;
function makestring(r : real; max, dec : integer) : bufstring;
function makestring2(r : real; max, dec : integer) : bufstring;
function makestringint(r : real) : bufstring;
procedure MC_rd(s : bufstring);
function mcs_init(w : integer) : boolean;
procedure MoveCursor(axis, numberofpoints : integer);
procedure Mread(name : integer; numchars : integer);
{procedure Mscan;}
procedure Mwrite(name : integer; wbuf : bufstring; lfflag : boolean);
procedure NextFileNum(w : integer);
procedure NewCursor;
function NumptstoCode(r : real) : integer;
procedure Pause;
{procedure PickPeaks; }
function Power(number : real) : real;
function PowerOfTenPrefix(logpower : integer) : bufstring;
procedure PrintWave;
function PtoU(w, p, xaxismode : integer) : real;
function PtoV(w, p, xaxismode, yaxismode : integer) : real;
function PtoX(w, p : integer) : real;
function PtoY(w, p : integer) : real;
procedure rd_int(var v : integer; mess : bufstring; min, max : integer);
procedure rd_long(var v : longint; mess : bufstring; min, max : longint);
procedure rd_real(var v : real; mess : bufstring; min, max : real);
procedure rd_str(var v : bufstring; mess : bufstring);
procedure readroundandlimit(var r : real; min, max : real);
procedure Readln2i(var i : integer);
procedure readln2l(var l : longint);
procedure Readln2r(var r : real);
procedure Readln2s(var i : bufstring);
procedure readvalue(var r : real; p : real);
function ReadWave(fn : bufstring; waveindex : integer; userinput :
  boolean) : boolean;
procedure ReadWaveE(var f : text; w : integer);
procedure ReadWaveEUpgrade(w : integer);
procedure ReadWaveF(var f : text; w : integer);
procedure ReadWaveFUpgrade(w : integer);
procedure ReadWaveG(var f : text; w : integer);
procedure ReadWaveGUpgrade(w : integer);
procedure ReadWaveH(var f : text; w : integer);
procedure ReadWaveHUpgrade(w : integer);
procedure ReadWaveI(var f : text; w : integer);
procedure ReadWaveIUpgrade(w : integer);
procedure ReadWaveJ(var f : text; w : integer);
procedure ReadWaveJUpgrade(w : integer);
procedure ReadWaveK(var f : text; w : integer);
procedure rebin;
procedure _rescale;
function rm_extension(s : bufstring) : bufstring;
function rounddown(r : real) : longint;
procedure roundoff(var r : real; s : real);
function roundup(r : real) : longint;

implementation

uses
  crt, dos, FpesCom, FpesAI, FpesST, FpesUZ, graph, Keys, TPDecl;

procedure limit(var r : real; min, max : real);
{ Constrains r to lie between min and max limits. }
begin
  if r < min then
    r := min
  else if r > max then
    r := max;
end;

function Log10(r : real) : real;
{ Calculates common logarithm of r. }
begin
  if (r <= 0 ) then
    Log10 := ln(-r) / LN_10
  else
    Log10 := ln(r) / LN_10;
end;

procedure MakeFilter(res : real);
{ Sets up Lorentzian filter function with FWHM of res. }
var
  i : integer;
begin
  res := 4 / (res * res); { Put in convenient form for calculation. }
  for i := 1 to MAXPOINTS div 2 + 1 do
    filter[i] := 1 / (1 + (i - 1) * (i - 1) * res);
end;

procedure MakeLookup;
{ Creates lookup table (ts.lookup) using wob parameters. Form of
  function is a single cycle of a sine wave added to a line with eq.
  y = x. Phase is not considered until the StageLookup function. }
var
  i : integer;
begin
  for i := 0 to TS_LOOKUP_MAX do
    ts.lookup[i] := i * ts.wob.per / TS_LOOKUP_MAX +
      sin(i * TWOPI / TS_LOOKUP_MAX) * ts.wob.ampl;
```

```
end;

function makestring(r : real; max, dec : integer) : bufstring;
{ Converts real number r into string with minimum possible length, but having
  at least dec decimal places. However, overall length is kept to <= max digits
  with truncation occurring from left, so decimal places are not guaranteed. }
var
  s : bufstring;
begin
  str(r : 0 : dec, s);
  if length(s) > max then
    makestring := copy(s, 1, max)
  else
    makestring := s;
end;

function makestring2(r : real; max, dec : integer) : bufstring;
{ Converts r into string having exactly max characters, and dec decimal
  places; extra characters are padding added on the left. Assumes r has
  been divided by appropriate power already so that there are no more
  than 3 digits to left of decimal point. Note that routine only really
  works if dec > 3; otherwise, for instance, if dec = 3, will encounter
  situation where r / p > 100 and so truncation will kill first fraction-
  al number but not decimal point. }
var
  s : bufstring;
begin
  str(r : 0 : dec, s);
  com_wr_db('"' + s + '"');
  if (r > -1) and (r < 0) then
    s := copy(s, 1, dec + 3) { 3 extra chars: -0.XXX }
  else if r < 1 then
    s := copy(s, 1, dec + 2) { 2 extra chars: -X.XX or 0.XXX }
  else
    s := copy(s, 1, dec + 1); { 1 extra char: X.XX }
  if max > length(s) then
    makestring2 := copy(BLANKLINE, 1, max - length(s)) + s
  else
    makestring2 := s;
end;

function makestringint(r : real) : bufstring;
{ Converts r into integer string. }
var
  s : bufstring;
begin
  str(r : 0 : 0, s);
  makestringint := s;
end;

procedure MC_rd(s : bufstring);
{ Read in mc file with name s. }
var
  c : char; { Dummy char to handle space between # and string. }
  f : text;
begin
{  if not FullPath(s) then
    s := dir + s; }
  if mc.fn = '' then
  begin
    mc.num := 0;
    exit;
  end;
  if not FileExists(s) then
  begin
    com_wr(s + ' not found.', COLORHL);
    mc.fn := '';
    mc.num := 0;
    exit;
  end;
  mc.fn := s;
  assign(f, s);
  reset(f);
  mc.num := 0;
  while (not eof(f)) and (mc.num < mc_MAX) do
  begin
    inc(mc.num);
    with mc.ls[mc.num] do
      readln(f, m, min, max, c, s); { Read mass, max and label. }
  end;
  close(f);
end;

function mcs_init(w : integer) : boolean;
{ Set up MCS for scanning, using abbreviated command sequence if para-
  meters in wave w match those currently in use. If w = 0, this is a
  flag to use the tw (tweak) variables instead, and adds a few more
  commands particular to the tw mode. Returns false only if MCS was
  not found on the GPIB bus. }
{ 3/18/97: Added communication with Tektronix 744A scope as well, for
  implementation of STS (shot-to-shot) mode. }
var
  devname : nbuf;
  j : integer;
  s, s2 : bufstring;
begin
  mcs_init := true;
  if mcs.new = 1 then
  begin
    { Check that MCS is on GPIB card. }
    devname := 'mcs    ';
    mcs.addr := ibfind(devname);
    mwrite(mcs.addr, 'clrs', false); { Clear device. }
    if (ibsta and ERR) <> 0 then
    begin
      error;
      mcs_init := false;
      exit;
    end;
    mwrite(mcs.addr, 'outp 1', false); { Direct device output to gpib
      card. }
    mwrite(mcs.addr, 'trlv .5', false); { Trigger level. }
    mwrite(mcs.addr, 'trsl 0', false); { Trigger slope = positive. }
    str(-discrim, s);
    mwrite(mcs.addr, 'dclv ' + s, false); { Discrimination level. }
    mwrite(mcs.addr, 'dcsl 1', false); { Discrimination slope = neg. }
    mwrite(mcs.addr, 'bckl 0', false); { Internal clock timebase. }
    mwrite(mcs.addr, 'left 0', false); { Set leftmost bin to 0. }
    mwrite(mcs.addr, 'botm 0', false); { Set bottom bin to 0 cts. }
    if w > 0 then { Normal mode. }
      with wv.ls[w]^ do
      begin
        mcs.pt := par.pt;
        mcs.shotsperscan := par.ele.shotsperscan;
        mcs.timeperpt := par.timeperpt;
        { Undo tweak mode settings. }
```

```
      mwrite(mcs.addr, 'hscl 7', false); { Set 1024 bins per screen.}
    end
  else
  begin { Tweak mode. }
    mcs.pt := tw.pt;
    mcs.shotsperscan := tw.shotsperscan;
    mcs.timeperpt := tw.timeperpt;
    { Additional commands for tweak mode. }
    mwrite(mcs.addr, 'hscl 4', false); { Set bins per screen to 128.}
{    mwrite(mcs.addr, 'llim 0', false); { Set left limit for integra-
      tion to bin 0. }
{    mwrite(mcs.addr, 'rlim 127', false);
      { Set right limit to 127 bins. }
  end;
  mwrite(mcs.addr, 'brec ' + makestringint(NumptsToCode(mcs.pt)),
    false); { Bins per record. }
  mwrite(mcs.addr, 'rscn ' + makestringint(mcs.shotsperscan), false);
    { Records per scan. }
  mwrite(mcs.addr, 'bwth ' + makestringint(TimebaseToCode(mcs.
    timeperpt)), false); { Select bin width. }
  delay(250); { Give it time. }

  { STS: Prepare TEK (except in tweak mode). }
  if (w > 0) and (_bs.sts = 1) then
  begin
    { Check that TEK is on GPIB card. }
    devname := 'tek    ';
    tek := ibfind(devname);
    if tek < 0 then
    begin
      com_wr('Tek find error.', COLORHL);
      mcs_init := false;
      exit;
    end;
    ibclr(tek);
    { Shut off possible acquisition in progress (don't use
      tekwrite since this waits for scope to be free first,
      which defeats purpose of shutting off an acquisition!) }
{    tekset_rd(_bs.sts_ch); { Record settings before changing. }
    mwrite(tek, 'acq:state 0', true);
    { No zoom: }
    tekwrite('zoom:state off');
    { Disable fit to screen: }
    tekwrite('hor:fittoscreen off');
    { Average multiple scans: }
    tekwrite('acq:mode ave');
    { Number of shots to average: }
    str(wv.ls[_bs.bg]^.par.ele.shotsperscan, s);
    tekwrite('acq:numavg ' + s);
    { Stop after set number of scans: }
    tekwrite('acq:stopafter seq');
    { Set up string for use by channel commands: }
    s := 'ch' + chr(ord('0') + _bs.sts_ch);
    { Set channel for data acquisition: }
    tekwrite('data:source ' + s);
    { Turn on channel: }
    tekwrite('sel:' + s + ' on');
    { Set vert. position to 0. }
    tekwrite(s + ':pos 0');
    { Set vert. scale. }
    str(_bs.sts_vert, s2);
    tekwrite(s + ':scale ' + s2);
    { Read vert. scale, store in _bs.sts_vert. }
{    tekwrite(s + ':scale?');
    mread(tek, sizeof(s2));
    s2 := '';
    for j := 1 to ibcnt - 1 do
      s2 := s2 + ibbuf[j];
    val(s2, _bs.sts_vert, j); { j is dummy variable. }
    { Set vert. offset to -5 div - discrim: }
    str(-5 * _bs.sts_vert - discrim, s2);
    tekwrite(s + ':offset ' + s2);
    { 2 bytes per bin: }
    tekwrite('data:width 2');
    { Transfer desired number of data points: }
    str(wv.ls[_bs.bg]^.par.pt, s);
    tekwrite('data:start 1;stop ' + s);
    { Use signed binary format with LSB (low byte) transferred
      first: }
    tekwrite('data:encdg sribinary');
    { Calculate minimum # of bins and set: }
    j := round(wv.ls[_bs.bg]^.par.pt);
    if j <= 500 then
      j := 500
    else if j <= 1000 then
      j := 1000
    else if j <= 2500 then
      j := 2500
    else
      j := 5000;
    str(j, s);
    tekwrite('hor:reco ' + s);
    { Set pretrigger to 0%: }
    tekwrite('hor:trig:pos 0');
    { Use main sweep: }
    tekwrite('hor:mode main');
    { Set time per division ( = time per pt * pts per div): }
    str(wv.ls[_bs.bg]^.par.timeperpt * TEK_PTPERDIV, s);
    tekwrite('hor:main:scale ' + s);
  end;
  mcs.new := 0;
end

{ When mcs.new = 0 ... }
else if w > 0 then
  { Selective commands sent if parameters have changed (don't bother
    for w = 0 case, since never more than one tweak going at once!) }
  with wv.ls[w]^ do
  begin
    if mcs.timeperpt <> par.timeperpt then
    begin
      mcs.timeperpt := par.timeperpt;
      mwrite(mcs.addr, 'brec ' + makestringint(NumptsToCode(mcs.pt)),
        false); { Bins per record. }
    end;
    if mcs.shotsperscan <> par.ele.shotsperscan then
    begin
      mcs.shotsperscan := par.ele.shotsperscan;
      mwrite(mcs.addr, 'rscn ' + makestringint(mcs.shotsperscan), false);
        { Records per scan. }
    end;
    if mcs.timeperpt <> par.timeperpt then
    begin
      mcs.timeperpt := par.timeperpt;
      mwrite(mcs.addr, 'bwth ' + makestringint(TimebaseToCode(mcs.
        timeperpt)), false); { Select bin width. }
```

```
      end;
    end;

  { Commands sent every time. }

  { STS mode: Start the TEK. }
  if (w > 0) and (_bs.sts = 1) then
    tekwrite('acq:state run');

  { STS mode (bs set only) : run toggle mode. }
  if (w > 0) and (_bs.sts = 1) and (wv.ls[w]^.par.ele.bs.mode = 1) and
    (_bs.sts_tog = 1) then
    mwrite(mcs.addr, 'acmd 2', false)
  else
    mwrite(mcs.addr, 'acmd 0', false);

  { Start the MCS. }
{  mwrite(mcs.addr, 'clrs', false); { Clear device twice for good measure. }
  mwrite(mcs.addr, 'clrs', false);
  mwrite(mcs.addr, 'sscn', false); { Start the scan. }
  mwrite(mcs.addr, 'locl 0', false); { Allow front panel to remain
    active while scanning. }

  if w > 0 then
    with wv.ls[w]^ do
    begin
      Time(scan.starttime); { Record start time. }
      scan.steptime := 100 * round(mcs.shotsperscan / par.ele.
        reprate); { How long to wait between data reads. }
    end
  else
  begin
    Time(tw.starttime);
    tw.steptime := 100 * round(mcs.shotsperscan / tw.reprate);
  end;
end;

procedure MoveCursor(axis, numberofpoints : integer);
{ Two functions:
  1. If xh.mode = 0: For axis = 0 (x), moves cursor a specified number
  of points along a wave (use numberofpoints = 0 when changing wave only). For
  axis = 1 (y), calls ChangeCurrentWave.
  2. If xh.mode = 1: Moves active crosshair corresponding to axis (0 =
  x, 1 = y) and numberofpoints. }
begin
  with sc.ls[sc.cur].gr do
    if xh.mode = 0 then
    begin
      if wv.num = 0 then
        exit;
      if (axis = 0) and (wv.ls[wv.cur]^.screen = sc.cur) then { Must be
        left/right arrow and wave on current screen in order to move. }
      begin
        if cursorvisible = 1 then
          EraseCursor(sc.cur)
        else
        begin
          cursoru := (u1lim + u2lim) / 2;
          UpdateCursor(sc.cur);
        end;
        { Calculate new cursor position. }
        if (xaxismode = XAXISMODE_CONVERT) and (xaxisdt =
          dt_ELE) then
          cursoru := PtoU(wv.cur, cursorp - numberofpoints, xaxismode)
          { Go backward for energy. }
        else
          cursoru := PtoU(wv.cur, cursorp + numberofpoints, xaxismode);
        { Make sure cursor is on visible part of x axis; calculate screen coordi-
        nates. }
        UpdateCursor(sc.cur);
        DrawCursor(sc.cur);
        DrawCursorInfo;
        if (mc.auto = 1) and (mc.s <> '') then
          com_wr(mc.s, COLORMESS);
      end;
    end
    else with xh do
    begin
      EraseCursor(sc.cur);
      if axis = 0 then
      begin
        x[which] := x[which] + numberofpoints;
        if x[which] < plotarea.x1 then
          x[which] := plotarea.x1
        else if x[which] > plotarea.x2 - 1 then
          x[which] := plotarea.x2 - 1;
      end
      else
      begin
        y[which] := y[which] + numberofpoints;
        if y[which] < plotarea.y1 then
          y[which] := plotarea.y1
        else if y[which] > plotarea.y2 - 1 then
          y[which] := plotarea.y2 - 1;
      end;
      UpdateCursor(sc.cur);
      DrawCursor(sc.cur);
      DrawCursorInfo;
    end;
end;

procedure mread(name : integer; numchars : integer);
{ Reads a character array from the device name and changes it to a
  string array rdbuf. }
var
  i : integer;
begin
  rdbuf := '';
  ibrd(name, ibbuf, numchars); { Numchars is a guess to how long the
    string is. }
  rdbuf[0] := chr(ibcnt - 1); { The length of rdbuf is set to the number
    of characters read, ibcnt - 1. }
  if (ibsta and ERR) <> 0 then error;
  for i := 1 to ibcnt do
    rdbuf[i] := ibbuf[i];
end;

{procedure Mscan;

const
  MaxIbbuf = $1000;
type
  iobuf = array[1..MaxIbbuf] of char;
  iolbuf = array[1..200] of integer;
var
    ibbuf : iobuf;
```

```
  devname : nbuf;
  hbuf : iolbuf;
  i,j,k,scan,result,terr : integer;
  stringconv,c,s,bina : bufstring;
  c2,numblocks : longint;
begin
  if wv.ls[wv.cur]^.saved = 0 then ( If there is already data in wave
    make sure users wants to overwrite it. )
 {   begin
     message := 'Overwrite Existing Wave?';
     DrawMessageBox;
     c := readkey;
     if (c = #89) or (c = #121) then  { 'y' or 'Y' )
 {       begin
         message := 'Overwriting';
         DrawMessageBox;
         delay(1000);
       end
      else
        begin
          message := 'Scan Aborted.';
          DrawMessageBox;
          delay(1000);
          message := MESSAGEDEFAULT;
          DrawMessageBox;
          exit;
        end;
    end;
  message := 'Starting Mscan';
  DrawmessageBox;
  FillChar(wv.ls[wv.cur]^.data,sizeof(wv.ls[wv.cur]^.data),0);      { Initalize
arrays. }
{  devname := 'magic  ';
  magic := ibfind(devname);    { Locate device on gpib card. }
{  if (ibsta and err) <> 0 then error;
  if magic < 0 then error;
  ibclr(magic);
  delay(200);
  ibclr(magic);
  delay(200);
  message := 'Found Magic Controller';
  DrawMessageBox;
  mwrite(magic,'ECHO OFF ');
  mwrite(magic,'A 0 ');
  mwrite(magic,'N 18 ');
  mwrite(magic,'Z ');
  mwrite(magic,'F 26 ');
  mwrite(magic,'E ');
  mwrite(magic,'COMM_FORMAT OFF,WORD,BINARY ');
  mwrite(magic,'COMM_BLOCKSIZE 200 ');
  mwrite(magic,'COMM_ORDER LOFIRST ');
  delay(100);                                  {Ridiculous but have to have it.}
{  mwrite(magic,'USING 18 ');
  mwrite(magic,'AVGENABLE ON');
  delay(100);                                  {Same}
{  mwrite(magic,'W 255 ');                        {set offset  }
{  mwrite(magic,'F 19 ');
  mwrite(magic,'E ');
  str(256 + 16*timebasetocode(wv.ls[wv.cur]^.timebase),stringconv);
  mwrite(magic,'W ' + stringconv + ' ');
  delay(100);
  mwrite(magic,'F 16 ');
  mwrite(magic,'E ');
  str(wv.ls[wv.cur]^.numpts+10,stringconv);
  mwrite(magic,'AVGLEN ' + stringconv + ' ');
  str(wv.ls[wv.cur]^.numscans,stringconv);
  mwrite(magic,'AVGN ' + stringconv + ' ');
  mwrite(magic,'AVG START ');
  delay(100);
  message := 'Started MScan';
  DrawMessageBox;
  repeat
   result := 0;
   mwrite(magic,'AVGDONE ');
   mread(magic,20);
   terr := 0;
   char_to_int(result,terr);
      if keypressed then
        begin
          c := readkey;
          if c = #27 then
            begin
              message := 'Scan Aborted';
              DrawMessageBox;
              delay(2000);
              message := MESSAGEDEFAULT;
              DrawAll;
              exit;
            end;
        end;
      delay(1000);
  until result > 0;
  message := 'About to read.';
  Drawmessagebox;
    numblocks := round(wv.ls[wv.cur]^.numpts/100);
    str(round(200*numblocks-1),stringconv);
    mwrite(magic,'READ -A,0,' + stringconv + ' ');
    delay(100);
    message := 'dataread sent';
    drawmessagebox;
{    for j := 0 to round(2*wv.ls[wv.cur]^.numpts/200) do
      begin
        dataread(magic,200);
        for i := 1 to 200 do
          wv.ls[wv.cur]^.scan[10*j+i] :=  wv.ls[wv.cur]^.scan[i];
      end;        }
{      textmode;
      scan.w := wv.cur;
     for j := 0 to numblocks-1 do
       begin
       writeln('going into dataread # ',j);
       delay(500);
         dataread(magic,200);
         for i := 1 to 100 do
           wv.ls[wv.cur]^.data[j*100+i] := scan.data[i];
       end;
       graphicsmode;
{    for j := 0 to numblocks-1 do
      begin
        ibrd(magic,ibbuf,200);
        message := 'ibrding';
        drawmessagebox;
        for k := 1 to 200 do
          wv.ls[wv.cur]^.data[k+100*j] := wv.ls[wv.cur]^.data[k+100*j] +
            wv.ls[wv.cur]^.scan[k];
      end;        }
```

```
{  wv.ls[wv.cur]^.saved := 0;
  FindLargest(wv.cur);
  FullView;
end; }

procedure mwrite(name : integer; wbuf : bufstring; LFFLAG : boolean);
{ Changes string to character array which is sent to the
  device, mcs.addr. }
const
  LF = #10;
var
  wlen, i : integer;
begin
  wlen := length(wbuf);
  for i := 1 to wlen do
    ibbuf[i] := wbuf[i];
  if lfflag then
  begin
    ibbuf[wlen + 1] := LF;
    inc(wlen);
  end;
  ibwrt(name, ibbuf, wlen);
  for i := 1 to wlen do
    ibbuf[i] := ' ';
  if (ibsta and ERR) <> 0 then
    com_wr('Error sending "' + wbuf + '"', COLORHL);
end;

procedure NextFileNum(w : integer);
{ Finds current file number and increments. If file number cannot be
  found (invalid fn), sets filename to ''. }
var
  dummy : integer;
  ex : bufstring;
  i : integer;
  s, s2 : bufstring;
  v : integer;
begin
  with wv.ls[w]^.par do
  begin
    { If using standard file extension, change to reflect dt (otherwise,
      keep intact). }
    ex := get_extension(fn);
    for i := dt_MIN to dt_MAX do
      if ex = dt_NAME[i] then
        ex := dt_NAME[dt];

    { Find current file number. }
    s := rm_extension(fn);
    i := length(s);
    while (i > length(s) - 4) and (i > 0) and (s[i] <> '-') do
      dec(i);
    if (i = 0) or (s[i] <> '-') then
    begin
      fn := '';
      exit;
    end;

    { Compare root with auto.ser, updating auto.num if they match and
      auto.num < v. }
    val(copy(s, i + 1, length(s) - i), v, dummy);
    s := copy(s, 1, i);
    if not FullPath(s) then
      s := dir + s;
    inc(v);
    s2 := auto.ser;
    if not FullPath(s2) then
      s2 := dir + s2;
    if (s = s2) and (auto.num <= v) then
      auto.num := v + 1;

    { Update filename. }
    fn := s + makestringint(v) + '.' + ex;
  end;
end;

procedure NewCursor;
{ Place cursor in middle of wave wv.cur. }
begin
  if wv.num = 0 then
    exit;
  with sc.ls[wv.ls[wv.cur]^.screen].gr do
    cursoru := (PtoU(wv.cur, wv.ls[wv.cur]^.pmin, xaxismode) +
      PtoU(wv.cur, wv.ls[wv.cur]^.par.pt, xaxismode)) / 2;
end;

function NumptstoCode(r : real) : integer;
{ Converts the numpts number to the corresponding mcs integer. }
var
  i : integer;
begin
  for i := MCS_PT_MIN to MCS_PT_MAX do
    if r = MCS_PT[i] then
    begin
      NumptsTocode := i;
      exit;
    end;
  NumptsToCode := -1;  { If something doesn't work, assigns -1. }
end;

procedure Pause;
{ Suspend operations until user presses a key. }
begin
  com_wr('Paused. Press any key.', COLORHL);
  { Read key, and if extended code, read next char as well. }
  if readkey = EXTENDED then
    readkey;
end;

(procedure PickPeaks;
var
  peak : array [1..MAXPEAKS] of integer;
  after : real;
  before : real;
  ch : char;
  count : integer;
  i,j : integer;
  message : bufstring;
  modified : boolean;
  ok : boolean;
  s : bufstring;
  temp : real;
  w : integer;

begin
  if (wv.num > 0 ) then
```

```
  begin
    for i:=1 to MAXPEAKS do
      peak[i] := 0;        { reset values }
{    textmode;
    writeln(' ');
    writeln('Procedure for finding peaks in a spectrum.');
    writeln(' ');
    repeat
      modified := FALSE;
      writeln('Present peak threshold is  : ',peakthreshold:5:3,'.');
      writeln('Present 2nd deriv. term is : ',peaksecond:6:3,'.');
      writeln(' ');
      writeln('Modify these (Y/N)?');
      repeat
        ch := Upcase(Readkey);
      until (ch in ['Y','N']);
      writeln(ch);
      if (ch = 'Y') then
        begin
          writeln(' ');
          writeln('New value for peak threshold (default =
',peakthreshold:5:3,').');
          readln(s);
          if (s <> '') then
            begin
              modified := TRUE;
              val_r(s,temp);
              if ((temp < 0.0) or (temp > 1.0)) then
                writeln('Value must be a percentage from 0 to 1')
              else
                peakthreshold := temp;
            end;
          writeln('New value for second deriv. (default = ',peaksecond:6:3,').');
          readln(s);
          if (s <> '') then
            begin
              modified := TRUE;
              val_r(s,temp);
              if (temp > 0.0) then
                temp := temp * -1.0; { peaksecond should be negative }
{             peaksecond := temp;
            end;
      end; { if }
{   until (modified = FALSE);
    writeln(' ');
    repeat
      ok := TRUE;
      writeln('Check for peaks in which wave? (default = ',wv.cur,').');
      readln(s);
      val_i(s,w);
      if ((w < 0) or (w > wv.num)) then
        begin
          ok := FALSE;
          writeln('Wave number out of range.');
        end;
    until (ok = TRUE);
    if (s = '') then
      w := wv.cur;
    count := 0;
    message := '';
    with wv.ls[w]^ do
      begin
        for i := pmin+1 to numpts-1 do
          begin
            { compute second derivative)
{           temp := (PtoV(w,i-1) + PtoV(w,i+1) - 2.*PtoV(w,i))/sqr(timebase);
            before := PtoV(w,i) - PtoV(w,i-1);
            after := PtoV(w,i+1) - PtoV(w,i);
            if (( temp <= peaksecond ) and
                ( before >= 0.0 ) and
                ( after <= 0.0 ) and
                ( PtoV(w,i)/largest >= peakthreshold )) then
              { found one }
{             begin
                inc(count);
                if (count<=MAXPEAKS) then
                  peak[count] := i
                else
                  { limit exceeded }
{                 begin
                    str(MAXPEAKS,s);
                    message :=
                    'Peak limit exceeded.  Maximum number of peaks is ' + s;
                  end;
              end;
          end;
      end;
    writeln('***** FINISHED *****');
    writeln('Found ',count,' peaks.');
    writeln(' ');
    writeln(message);
    waitkey;
    if (count > MAXPEAKS ) then
      count := MAXPEAKS;
    if (count <> 0 ) then
    begin
      with wv.ls[w]^ do
      begin
        case scantype of
          TIMESCAN,LONGTIMESCAN:
          begin
            j := 0;
            repeat
              clrscr;
              writeln(' ');
              writeln('PEAK LIST FOR WAVE NUMBER ', w, ' ('+wavename+').');
              writeln(' ');
              writeln('Peak   Time(ns)   Rel. Int.   Abs. Int.  Sec. Deriv.');
              writeln('----   --------   ---------   ---------  -----------');
              i := 0;
              while (i < LINESPERPAGE) do
                begin
                  inc(i);
                  if ((i+j <= MAXPEAKS) and (i+j <= count)) then
                    writeln((i+j):4,'  ', (PtoU(w,peak[i+j],xaxismode)):9:0,'  ',
                      (AbstoRel(w,PtoV(w,peak[i+j],xaxismode,yaxismode))):10:3,'  ',
                      (PtoV(w,peak[i+j],xaxismode,yaxismode)):10:0,'    ',
                      ((PtoV(w,peak[i+j]+1,xaxismode,yaxismode) + PtoV(w,peak[i+j]-
1)
                      -2.0*PtoV(w,peak[i+j]))/sqr(timebase)):10:3);
                end;
              j := j + LINESPERPAGE;
              waitkey;
            until (j >= count);
          end; { TIMESCAN, LONGTIMESCAN }
{         ENERGYSCAN:
```

```
          begin
            j := 0;
            repeat
              clrscr;
              writeln(' ');
              writeln('PEAK LIST FOR WAVE NUMBER ', w, ' ('+wavename+').');
              writeln(' ');
              writeln('Peak   Time(ns)  Energy (eV)   Rel. Int.   Abs. Int.  Sec.
Deriv.');
              writeln('----   --------  -----------   ---------   ---------  -------
----');
              i := 0;
              while (i < LINESPERPAGE) do
                begin
                  inc(i);
                  if ((i+j <= MAXPEAKS) and (i+j <= count)) then
                    writeln((i+j):4,'  ',
                      (peak[i+j]*timebase + delaytime):9:0, '  ',
                      (PtoU(w,peak[i+j])):11:3,'  ',
                      (AbstoRel(w,PtoV(w,peak[i+j]))):10:3,'  ',
                      (PtoV(w,peak[i+j])):10:0,'   ',
                      ((PtoV(w,peak[i+j]+1) + PtoV(w,peak[i+j]-1)
                     -2.0*PtoV(w,peak[i+j]))/sqr(timebase)):10:3);
                end;
              j := j + LINESPERPAGE;
              waitkey;
            until (j >= count);
          end; { ENERGYSCAN }
{       MASSSCAN:
          begin
            j := 0;
            repeat
              clrscr;
              writeln(' ');
              writeln('PEAKLIST FOR WAVE NUMBER ', w, ' ('+fn+').');
              writeln(' ');
              writeln('Peak   Time(ns)    Mass (Da)   Rel. Int.   Abs. Int.  Sec.
Deriv.');
              writeln('----   --------    ---------   ---------   ---------  -------
----');
              i := 0;
              while (i < LINESPERPAGE) do
                begin
                  inc(i);
                  if ((i+j <= MAXPEAKS) and (i+j <= count)) then
                    writeln((i+j):4,'  ',
                      (peak[i+j]*timebase + delaytime):9:0,'  ',
                      (PtoU(w,peak[i+j])):11:2,'  ',
                      (AbstoRel(w,PtoV(w,peak[i+j]))):10:3,'  ',
                      (PtoV(w,peak[i+j])):10:0,'   ',
                      ((PtoV(w,peak[i+j]+1) + PtoV(w,peak[i+j]-1)
                     -2.0*PtoV(w,peak[i+j]))/sqr(timebase)):10:3);
                end;
              j := j + LINESPERPAGE;
              waitkey;
            until (j >= count);
          end; { MASSSCAN }
{       end; { case }
{     end; { with }
{   end; { if }
{   DrawAll;
  end; { if }
{end;}
```

```
function Power(number : real) : real;
{ Calculates power of 1000 which, when divided from number, leaves a mantissa
  between 1 and 999.9... If number = 0, Power returns 1. }
begin
  if number = 0 then
    Power := 1
  else
    Power := XToTheY(1000, rounddown(Log10(number) / 3 + SMALL)); { SMALL is
    added to force function to advance to next power of 1000 prematurely.
    This is to ensure numbers are less than 1000 (though there is a chance
    they may be less than 1). Preference is for an interval to be 0..0.9
    rather than 0..900, especially on vertical axis when displaying
    normalized spectra. }
end;

function PowerOfTenPrefix(logpower : integer) : bufstring;
{ Like UnitPrefix(), but produces notation like '10^-3' rather than SI unit
  prefixes. If logpower = 0, returns ''. }
begin
  if logpower = 0 then
    PowerOfTenPrefix := ''
  else
    PowerOfTenPrefix := '10^' + makestring(logpower, 4, 0); { Extra
    space added for clarity with units succeeding, i.e., '10^3 eV'. I use 4
    digits since this provides room for up to 10^-999, adequate to indicate
    if there's a problem; if only 3 digits used, will truncate to left, so
    that logpower = -100 turns into -10, not a good indicator! }
end;

procedure PrintWave;
var
  user : text;
  i : integer;
  s : bufstring;
begin
  if wv.num = 0 then
    exit;
  s := wv.ls[wv.cur]^.par.fn;
  i := FilenameStart(s);
  if i = 1 then
    s := printdir + 'P' + s
  else
    s := copy(s, 1, i - 1) + 'P' + copy(s, i, length(s));
  if FileExists(s) then
    if com_wr_yn('File exists. Overwrite') = 0 then
      exit;
  if FileOpenWrite(user, s) = false then
    exit;
  com_wr('Writing print file.', COLORMESS);
  writeln(user, wv.ls[wv.cur]^.par.fn);
  for i:= 1 to wv.ls[wv.cur]^.par.pt do
    writeln(user, wv.ls[wv.cur]^.data[i]);
  close(user);
end;

function PtoU(w, p, xaxismode : integer) : real;
{ Converts point p in wave w to unit space x coordinate for any xaxismode
  (can specify mode other than current one). U depends both on xaxismode and
  dt of wave:

  xaxismode          wave[w]^.par.dt   x axis dimensions
  -----------------  ---------------   -----------------
```

```
  XAXISMODE_POINTS    (any)            points
  XAXISMODE_NORMAL    dt_COR           distance (um)
  "                   dt_ELE           time (ns)
  "                   dt_MAS           time (us)
  "                   dt_POW           time (ms to s)
  XAXISMODE_CONVERT   dt_COR           time (fs)
  "                   dt_ELE           energy (eV)
  "                   dt_MAS           mass (Da)
  "                   dt_POW           no change from XAXISMODE_NORMAL
}
var
  t : real; { Conversion variable for time -> energy. }
  a,b,c,invsqE,temp : real; { temporary variables }
begin
  if p < 1 then
  begin
    ptou := 0;
    exit;
  end;
  with wv.ls[w]^ do
    case xaxismode of
    XAXISMODE_POINTS:
      PtoU := p;
    XAXISMODE_NORMAL:
      case par.dt of
      dt_COR:
        PtoU := par.cor.ts.start + (p - 1) * par.cor.ts.step;
      dt_ELE, dt_POW:
        PtoU := p * par.timeperpt;
      dt_MAS:
        PtoU := p * par.timeperpt + par.mas.delay;
      end;
    XAXISMODE_CONVERT:
      case par.dt of
      dt_COR:
        PtoU := (par.cor.ts.start + (p - 1) * par.cor.ts.step
          - par.cor.ts.t0) / HALFSPEEDOFLIGHT;
      dt_ELE:
        if (par.ele.cal.quad = 0) then  { if quad cal factors are }
        begin                           { not being used, do normal }
          t := p * par.timeperpt - par.ele.cal.t0;
          PtoU := SLOPEFACTOR * sqr(par.ele.cal.len) / (t * t);
        end
        else
        begin
{         t := p * par.timeperpt;
{         if ( t < 0.2e-6 ) then        { My function doesn't work well }
{         begin                         { below 0.2us or above 4.8us, so }
{           PtoU := par.ele.cal.eV;     { fix these ranges so fxn doesn't crash }
{         end;
(1998-5-18 BJG: I need to access longer times -- comment this part:)
{         if ( t > 4.8e-6 ) then
          begin
            PtoU := 0.06;
          end; }
        {   if ( t >= 0.2e-6) and (t <= 4.8e-6 ) then
          begin }
            t := p * par.timeperpt - par.ele.cal.t0;
            a := par.ele.cal.quad * ELECTRONMASSKG / 2;
            b := sqrt(ELECTRONMASSKG/2) * (par.ele.cal.len -
              2 * par.ele.cal.quad * par.ele.cal.quadoff);
            c := par.ele.cal.quad * par.ele.cal.quadoff *
              par.ele.cal.quadoff - t;
            temp := b * b - 4 * a * c;
            if temp < 0 then
              temp := 0;
{           else
            begin }
              invsqE := (-b+sqrt(temp))/(2*a);
              if invsqE <> 0 then
                PtoU := (1 / (invsqE * invsqE)) * JTOEV
              else
                PtoU := 0;
{           end;    }
{         end; }
        end;
      dt_POW: { No change from XAXISMODE_NORMAL. }
        PtoU := p * par.timeperpt;
      dt_MAS:
        PtoU := par.mas.cal.sl * sqr(p * par.timeperpt + par.mas.delay
          - par.mas.cal.int);
      end;
    end;
end;

function PtoV(w, p, xaxismode, yaxismode : integer) : real;
{ Converts point p in wave w to unit space y coordinate, for any xaxismode/
  yaxismode combination (not limited to current ones). Two x axis units re-
  quire rescaling of y axis: energy and mass. Other units have the y axis
  passed through untouched.

  New addition: "display-only" background subtraction, that is, if
    selected (_bs.dis = 1), and w is an ele wave, PtoV will return
    the difference between wave w and the background wave _bs.bg,
    suitably scaled by the wave's bg.scale factor. }
var
  temp : real; { Temporary calculations. }
begin
  if p < 1 then
  begin
    ptov := 0;
    exit;
  end;
  with wv.ls[w]^ do
  begin
    if (par.dt = DT_ELE) and (xaxismode = XAXISMODE_CONVERT) then
    begin
      temp := PtoU(w, p, xaxismode);
      if temp > 0 then
        temp := data[p] / (temp * sqrt(temp)) { E ^ -1.5. }
      else
        temp := 0; { Trap to avoid imaginaries. }
    end
    else if (_bs.dis = 1) and (par.dt = DT_ELE) and (par.ele.bs.mode =
      1) and (_bs.bg <> w) and (_bs.bg > 0) and (wv.ls[_bs.bg]^.par.dt
      = DT_ELE) then
    { Handle bg subtraction display. }
    begin
      temp := wv.ls[_bs.bg]^.par.ele.bs.tot;
      if temp > 0 then
        temp := data[p] - wv.ls[_bs.bg]^.data[p] * par.ele.bs.tot /
          temp
      else
        temp := data[p];
    end
    else
```

```
      temp := data[p];
    if yaxismode = YAXISMODE_RELATIVE then
      PtoV := temp * abs(par.yscale) + par.yoffset
    else
      PtoV := temp;
  end;
end;

function PtoX(w, p : integer) : real;
{ Converts point p in wave w into a screen x coordinate. }
begin
  PtoX := UtoX(PtoU(w, p, sc.ls[wv.ls[w]^.screen].gr.xaxismode), wv.ls[w]^.
    screen);
end;

function PtoY(w, p : integer) : real;
{ Converts point p in wave w into a screen y coordinate. }
begin
  PtoY := VtoY(PtoV(w, p, sc.ls[wv.ls[w]^.screen].gr.xaxismode,
    sc.ls[wv.ls[w]^.screen].gr.yaxismode), wv.ls[w]^.screen);
end;

procedure rd_int(var v : integer; mess : bufstring; min, max : integer);
{ Either print current value of integer variable v (using message mess),
  or change to user's input provided it falls within range of min and
  max. }
var
  dummy : integer;
  u : integer;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    str(v, s);
    com_wr(mess + ' = ' + s, COLORMESS);
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], u, dummy);
  if (u >= min) and (u <= max) then
    v := u
  else
    com_err;
end;

procedure rd_long(var v : longint; mess : bufstring; min, max : longint);
{ Either print current value of longint variable v (using message mess),
  or change to user's input provided it falls within range of min and
  max. }
var
  dummy : integer;
  u : longint;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    str(v, s);
    com_wr(mess + ' = ' + s, COLORMESS);
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], u, dummy);
  if (u >= min) and (u <= max) then
    v := u
  else
    com_err;
end;

procedure rd_real(var v : real; mess : bufstring; min, max : real);
{ Same as rd_int, but for real values. }
var
  dummy : integer;
  u : real;
  s : bufstring;
begin
  if com.cur = com.num then
  begin
    str(v, s);
    com_wr(mess + ' = ' + s, COLORMESS);
    exit;
  end;
  inc(com.cur);
  val(com.ls[com.cur], u, dummy);
  if (u >= min) and (u <= max) then
    v := u
  else
    com_err;
end;

procedure rd_str(var v : bufstring; mess : bufstring);
{ Either print current value of string variable v (using message mess),
  or change to user's input. }
begin
  if com.cur = com.num then
  begin
    com_wr(mess + ' = "' + v + '"', COLORMESS);
    exit;
  end;
  inc(com.cur);
  v := com.ls[com.cur];
end;

procedure readroundandlimit(var r : real; min, max : real);
{ Read value into r from keyboard, round off, and constrain it within the
  limits min and max. }
begin
  readln2r(r);
  r := round(r);
  if r < min then
    r := min
  else if r > max then
    r := max;
end;

procedure readln2i(var i : integer);
{ Only changes the parameter value if a character is entered. }
var
  s : string[20];
  dummy : integer;
begin
  readln(s);
  if s<>'' then
   val(s, i, dummy);
end;

procedure readln2l(var l : longint);
```

```
{ Only changes the parameter value if a character is entered. }
var
  s : string[20];
  dummy : integer;
begin
  readln(s);
  if s<>'' then
   val(s, l, dummy);
end;

procedure readln2r(var r : real);
{ Only changes the parameter value if a character is entered. }
var
  s : string[20];
  dummy : integer;
begin
  readln(s);
  if s <> '' then
    val(s, r, dummy);
end;

procedure readln2s(var i : bufstring);
{ Only changes the parameter value if a character is entered. }
var s : bufstring;

begin
 readln(s);
 if s<>'' then
   i := s;
end;

procedure readvalue(var r : real; p : real);
{ Change value of r. If user hits return only, r is unchanged; otherwise,
  r is changed after multiplying by proper power conversion p. If p = 0,
  calculates appropriate power of p automatically. }
var
  dummy : integer; { Dummy variable needed by val. }
  s : bufstring; { Input string. }
begin
  if p = 0 then
    p := Power(r);
  s := '';
  readln(s);
  if s <> '' then
  begin
    val(s, r, dummy);
    r := r * p;
  end;
end;

function ReadWave(fn : bufstring; waveindex : integer; userinput :
  boolean) : boolean;
{ This procedure reads file fn into wv.ls[waveindex]. Must check that
  filename exists first. Data formats E and later are supported. Other
  wave parameters (yscale, etc.) are not changed. Userinput is a flag
  indicating whether to accept user input of additional parameters; if
  FALSE is passed, it is assumed that these parameters are already
  known (from ws_rd). }
var
  f : text;                          { File variable. }
  i : integer;                       { General counter. }
  ok : boolean;                      { Flag to indicate if load fails. }
  s : bufstring; { String variable for general use. }
begin
  ok := TRUE;
  assign(f, fn);
  reset(f);
  readln(f, s); { Read first line to obtain version #. }
  if s[1] = 'E' then
    ReadWaveE(f, waveindex)
  else if s[1] = 'F' then
    ReadWaveF(f, waveindex)
  else if s[1] = 'G' then
    ReadWaveG(f, waveindex)
  else if s[1] = 'H' then
    ReadWaveH(f, waveindex)
  else if s[1] = 'I' then
    ReadWaveI(f, waveindex)
  else if s[1] = 'J' then
    ReadWaveJ(f, waveindex)
  else if s = HEAD_WV then { Current format. }
    ReadWaveK(f, waveindex)
  else
  begin
    com_wr('Data format not supported.', COLORHL);
    ok := FALSE;
  end;
  if ok then
    with wv.ls[waveindex]^ do
    begin
      par.fn := fn; { Assign filename. }
      for i := 1 to par.pt do { Read data points. }
        readln(f, data[i]);
    end;
  close(f);
  ReadWave := ok;

{ Preserve for posterity: oldest version formats: }
{    else { oldest format - no letter code. }
{      begin
        Reset(F);
        Readln(F, r); timebase := MCS_TIMEPERPT[round(r)];
        Readln(F, r); numpts := round(r);
        If SameScanType(waveindex) = FALSE then
          ok := FALSE
        else
        begin
          Readln(F, s);
          Readln(F, s);
          Readln(F, s);
        end;
        if userinput and ok and ((scantype = TIMESCAN) or (scantype =
          ENERGYSCAN)) then
        begin
          writeln('This file needs the following information provided:');
          writeln('  Laser wavelength');
          writeln('  Ion mass');
          writeln('  T0');
          writeln('  Length');
          writeln('  Float voltage');
          writeln('  Quadratic compression factor');
          writeln('If these quantities are readily available, press Y, or press any
other key to');
          writeln('exit.');
          case readkey of
          'y', 'Y': ; { Do nothing. }
```

```
{          else
             ok := FALSE;
           end;
           if ok then
           begin
             write('Laser wavelength (nm): ');
             readln(r);
             laserev := EVNM / r; { Convert wavelength to energy. }
{             write('Ion mass (Daltons): ');
             readln(ionmass);
             write('T0 (ns): ');
             readln(t0);
             write('Length (m): ');
             readln(leng);
             slope := SLOPEFACTOR * leng * leng;
             writeln('Float voltage (V): ');
             readln(float);
             delta_e := ELECTRONMASS * float / leng;
             write('Quadratic compression factor (ns.eV) (default = 0): ');
             readln(s);
             if s = '' then
               quad := 0
             else
               val_r(s, quad);
           end;
         end;
       end; }
{    writeln('File format: ' + s); }
{    'A':
       begin
         Readln(F, r); timebase := MCS_TIMEPERPT[round(r)];
         delaytime := 0;
         Readln(F, r); numpts := round(r);
         If SameScanType(waveindex) = FALSE then
           ok := FALSE
         else
         begin
           Readln(F, s);
           Readln(F, s);
           Readln(F, s);
           Readln(F, t0);
           Readln(F, leng); slope := SLOPEFACTOR * leng * leng;
         end;
         if userinput and ok and ((scantype = TIMESCAN) or (scantype =
           ENERGYSCAN)) then
         begin
           writeln('This file needs the following information provided:');
           writeln('  Laser wavelength');
           writeln('  Ion mass');
           writeln('  Float voltage');
           writeln('  Quadratic compression factor');
           writeln('If these quantities are readily available, press Y, or press any
other key to');
           writeln('exit.');
           case readkey of
           'y', 'Y': ; { Do nothing. }
{          else
             ok := FALSE;
           end;
           if ok then
           begin
             repeat
               found := TRUE;
```

```
               write('Laser wavelength (nm): ');
               readln(i);
               case i of
                 213 : p := 0;
                 266 : p := 1;
                 299 : p := 2;
                 355 : p := 3;
                 416 : p := 4;
                 532 : p := 5;
               else
                 found := FALSE
               end;
             until (found = TRUE);
             laserev := CODE_TO_LASEREV[p];
             write('Ion mass (Daltons): ');
             readln(IonMass);
             writeln('Float voltage (V): ');
             readln(float);
             delta_e := ELECTRONMASS * float / ionmass;
             write('Quadratic compression factor (ns.eV) (default = 0): ');
             readln(s);
             if s = '' then
               quad := 0
             else
               val_r(s, quad);
           end;
         end;
       end;
     'B':
       begin
         Readln(F, r); timebase := MCS_TIMEPERPT[round(r)];
         delaytime := 0;
         Readln(F, r); numpts := round(r);
         If SameScanType(waveindex) = FALSE then
           ok := FALSE
         else
         begin
           Readln(F, s);
           Readln(F, s);
           Readln(F, s);
           Readln(F, t0);
           Readln(F, leng); slope := SLOPEFACTOR * leng * leng;
           readln(f, quad);
         end;
         if userinput and ok and ((scantype = TIMESCAN) or (scantype =
           ENERGYSCAN)) then
         begin
           writeln('This file needs the following information provided:');
           writeln('  Laser wavelength');
           writeln('  Ion mass');
           writeln('  Float voltage');
           writeln('If these quantities are readily available, press Y, or press any
other key to');
           writeln('exit.');
           case readkey of
           'y', 'Y': ; { Do nothing. }
{          else
             ok := FALSE;
           end;
           if ok then
           begin
             repeat
               found := TRUE;
```

```
            write('Laser wavelength (nm): ');
            readln(i);
            case i of
              213 : p := 0;
              266 : p := 1;
              299 : p := 2;
              355 : p := 3;
              416 : p := 4;
              532 : p := 5;
            else
              found := FALSE
            end;
          until (found = TRUE);
          laserev := CODE_TO_LASEREV[p];
          write('Ion mass (Daltons): ');
          readln(ionmass);
          writeln('Float voltage (V): ');
          readln(float);
          delta_e := ELECTRONMASS * float / ionmass;
        end;
      end;
    end;
  'C':
    begin
      Readln(F, r); timebase := MCS_TIMEPERPT[round(r)];
      delaytime := 0;
      Readln(F, r); numpts := round(r);
      If SameScanType(waveindex) = FALSE then
        ok := FALSE
      else
      begin
        Readln(F, s);
        Readln(F, s);
        Readln(F, s);
        Readln(F, t0);
        Readln(F, leng); slope := SLOPEFACTOR * leng * leng;
        readln(f, quad);
        readln(f, s);
      end;
      if userinput and ok and ((scantype = TIMESCAN) or (scantype =
        ENERGYSCAN)) then
      begin
        writeln('This file needs the following information provided:');
        writeln('  Laser wavelength');
        writeln('  Ion mass');
        writeln('  Float voltage');
        writeln('If these quantities are readily available, press Y, or press any
other key to');
        writeln('exit.');
        case readkey of
        'y', 'Y': ; { Do nothing. }
{       else
          ok := FALSE;
        end;
        if ok then
        begin
          repeat
            found := TRUE;
            write('Laser wavelength (nm): ');
            readln(i);
            case i of
              213 : p := 0;
              266 : p := 1;
              299 : p := 2;
              355 : p := 3;
              416 : p := 4;
              532 : p := 5;
            else
              found := FALSE
            end;
          until (found = TRUE);
          laserev := CODE_TO_LASEREV[p];
          write('Ion mass (Daltons): ');
          readln(ionmass);
          writeln('Float voltage (V): ');
          readln(float);
          delta_e := ELECTRONMASS * float / ionmass;
        end;
      end;
    end;
  'D':
    begin { read all parameters, keeping only those needed. }
{     readln(f, s);
      readln(f, s);
      readln(f, r, s); numpts := round(r); { reading s after number skips text
        commentary }
{     readln(f, r, s); timebase := MCS_TIMEPERPT_D[round(r)];
      If SameScanType(waveindex) = FALSE then
        ok := FALSE
      else
      begin
        readln(f, s);
        readln(f, s);
        readln(f, s);
        readln(f, r, s); laserev := CODE_TO_LASEREV[round(r)];
        readln(f, r, s); ionmass := round(r);
        readln(f, t0, s);
        readln(f, leng, s); slope := SLOPEFACTOR * leng * leng;
        readln(f, float, s); delta_e := ELECTRONMASS * float / ionmass;
        readln(f, quad, s);
        readln(f, s);
        readln(f, s);
        readln(f, delaytime, s);
      end;
    end;)
end;

procedure ReadWaveE(var f : text; w : integer);
{ First FPES format (actually contained a lot of tenure parame-
  ters!), read protocol of which was revised 25JAN96 for compatibility
  with new data format 'G'. Parameters not contained in file are assigned
  their pardf[dt_ELE] value. }
var
  r : real;
  s : bufstring;
begin
  with wv.ls[w]^ do
  begin
    par := pardf[dt_ELE]; { Copy over defaults. }
    par.comment := 'Data format E (old FPES).';
    readln(f, s); { Skip wavename. }
    readln(f, s); { Skip 'update every.' }
    readln(f, r, s); par.pt_gl := round(r);
    readln(f, r, s); par.timeperpt := MCS_TIMEPERPT[round(r)];
    readln(f, s); { Skip discrimination level. }
    readln(f, s); { Skip time scan flag. }
```

```
    readln(f, s); { Skip background subtraction flag. }
    readln(f, s); { Skip laser energy. }
    readln(f, s); { Skip ion mass. }
    readln(f, r, s);
    par.ele.cal.t0 := r * 1e-9; { Convert ns -> s. }
    readln(f, r, s);
    par.ele.cal.len := r * 1e-2; { Convert cm -> m. }
    readln(f, s); { Skip float. }
    readln(f, s); { Skip quadratic compression factor. }
    readln(f, s); { Skip vertical scale. }
    readln(f, s); { Skip horizontal position. }
    readln(f, s); { Skip time offset. }
    readln(f, r, s); par.scan_gl := round(r);
    readln(f, s); { Skip timeFPES. }
  end;
  readwaveEupgrade(w);
end;

procedure ReadWaveEupgrade(w : integer);
begin
  ReadWaveFUpgrade(w);
end;

procedure ReadWaveF(var f : text; w : integer);
{ Added 25JAN96, this is a "screw-up" format which resembles E,
  except contains no parameters (just data), for files 148-6 thru 148-26
  only. }
begin
  with wv.ls[w]^ do
  begin
    par := pardf[dt_ELE]; { Copy over defaults. }
    par.comment := 'Note: Data format F (weird).';
  end;
  readwaveFupgrade(w);
end;

procedure ReadWaveFupgrade(w : integer);
begin
  ReadWaveGUpgrade(w);
end;

procedure ReadWaveG(var f : text; w : integer);
{ True new data format, 25JAN96. }
var
  r : real;
  s : bufstring;
begin
  with wv.ls[w]^ do
  begin
    readln(f, par.comment);
    readln(f, par.dt);
    readln(f, par.pt_gl);
    readln(f, par.yoffset);
    readln(f, par.yscale);
    case par.dt of
    dt_COR:
      begin
        readln(f, r); par.cor.ch := round(r);
        readln(f, r); par.scan_gl := round(r);
        readln(f, r); par.cor.shotsperpt := round(r);
        readln(f, par.cor.ts.start);
        readln(f, par.cor.ts.step);
        readln(f, par.cor.ts.stop);
        readln(f, par.cor.ts.t0);
        readln(f, par.timeperpt);
        readln(f); { Waittime -- now defunct. }
      end;
    dt_ELE:
      begin
        readln(f, par.ele.cal.len);
        readln(f); { Skip quad factor. }
        readln(f, par.ele.cal.t0);
        readln(f); { Skip counttotal. }
        readln(f, par.ele.cal.ev);
        readln(f, r); par.scan_gl := round(r);
        readln(f); { Skip powerpump }
        readln(f); { Skip powerprobe }
        readln(f, par.ele.reprate);
        readln(f, r); par.ele.shotsperscan := round(r);
        readln(f, par.ele.ts.pos);
        readln(f, par.ele.ts.t0);
        readln(f, par.ele.dly);
        readln(f, par.timeperpt);
      end;
    dt_POW:
      begin
        readln(f, r); par.pow.ch := round(r);
        readln(f, par.pow.cal.int);
        readln(f, par.pow.cal.sl);
        readln(f, par.timeperpt);
        par.scan_gl := pardf[dt_POW].scan_gl; { Not recorded,
          so make default. }
      end;
    end;
  end;
  ReadWaveGUpgrade(w);
end;

procedure ReadWaveGUpgrade(w : integer);
{ Modifies version G wave so is compatible with latest version. }
begin
  with wv.ls[w]^.par do
  begin
    alert := pardf[dt].alert;
    if dt = dt_ELE then
    begin
      ele.bs.tot := pardf[dt].ele.bs.tot;
      ele.bs.mode := pardf[dt].ele.bs.mode;
    end;
    sh := pardf[dt].sh;
    skip := pardf[dt].skip;
  end;
  ReadWaveHUpgrade(w);
end;

procedure ReadWaveH(var f : text; w : integer);
var
  s : bufstring;
begin
  with wv.ls[w]^ do
  begin
    readln(f, par.alert);
    readln(f, par.comment);
    readln(f, par.dt);
    readln(f, par.pt_gl);
    readln(f, par.scan_gl);
```

```
    readln(f, par.sh);
    readln(f, par.skip);
    readln(f, par.timeperpt);
    readln(f, par.yoffset);
    readln(f, par.yscale);
    case par.dt of
      dt_COR: with par.cor do
      begin
        readln(f, ch);
        readln(f, shotsperpt);
        readln(f, ts.start);
        readln(f, ts.step);
        readln(f, ts.stop);
        readln(f, ts.t0);
      end;
      dt_ELE: with par.ele do
      begin
        readln(f, s);
        readln(f, bs.tot);
        readln(f, bs.mode);
        readln(f, cal.ev);
        readln(f, cal.len);
        readln(f, cal.t0);
        readln(f, dly);
        readln(f, reprate);
        readln(f, shotsperscan);
        readln(f, ts.pos);
        readln(f, ts.t0);
      end;
      dt_POW: with par.pow do
      begin
        readln(f, ch);
        readln(f, cal.int);
        readln(f, cal.sl);
      end;
      dt_MAS: with par.mas do
      begin
        readln(f, cal.int);
        readln(f, cal.sl);
        readln(f, ch);
      end;
    end;
    readln(f); { Read blank line separating header from data. }
  end;
  ReadWaveHUpgrade(w); { Upgrade to latest version. }
end;

procedure ReadWaveHUpgrade(w : integer);
{ Modifies version H wave so is compatible with latest version. }
begin
  with wv.ls[w]^.par do
  begin
    vstop := pardf[dt].vstop;
    if dt = dt_MAS then
    begin
      mas.scantime := pardf[dt].mas.scantime;
      mas.vert := pardf[dt].mas.vert;
    end;
  end;
  ReadWaveIUpgrade(w);
end;

procedure ReadWaveI(var f : text; w : integer);
var
  s : bufstring;
begin
  with wv.ls[w]^ do
  begin
    readln(f, par.alert);
    readln(f, par.comment);
    readln(f, par.dt);
    readln(f, par.pt_gl);
    readln(f, par.scan_gl);
    readln(f, par.sh);
    readln(f, par.skip);
    readln(f, par.timeperpt);
    readln(f, par.vstop);
    readln(f, par.yoffset);
    readln(f, par.yscale);
    case par.dt of
      dt_COR: with par.cor do
      begin
        readln(f, ch);
        readln(f, shotsperpt);
        readln(f, ts.start);
        readln(f, ts.step);
        readln(f, ts.stop);
        readln(f, ts.t0);
      end;
      dt_ELE: with par.ele do
      begin
        readln(f, s);
        readln(f, bs.tot);
        readln(f, bs.mode);
        readln(f, cal.ev);
        readln(f, cal.len);
        readln(f, cal.t0);
        readln(f, dly);
        readln(f, reprate);
        readln(f, shotsperscan);
        readln(f, ts.pos);
        readln(f, ts.t0);
      end;
      dt_POW: with par.pow do
      begin
        readln(f, ch);
        readln(f, cal.int);
        readln(f, cal.sl);
      end;
      dt_MAS: with par.mas do
      begin
        readln(f, cal.int);
        readln(f, cal.sl);
        readln(f, ch);
        readln(f, delay);
        readln(f, inv);
        readln(f, scantime);
        readln(f, vert);
      end;
    end;
    readln(f); { Read blank line separating header from data. }
  end;
  ReadWaveIUpgrade(w);
end;

procedure ReadWaveIUpgrade(w : integer);
```

```
{ Modifies version I wave so is compatible with latest version. }
begin
  with wv.ls[w]^.par do
  begin
    gen := pardf[dt].gen;
    pt := pt_gl;
    scan := scan_gl;
    case dt of
      DT_COR:
        cor.ts.wob := pardf[dt].cor.ts.wob;
      DT_ELE:
        ele.ts.wob := pardf[dt].ele.ts.wob;
    end;
  end;
  ReadWaveJUpgrade(w);
end;

procedure ReadWaveJ(var f : text; w : integer);
var
  s : bufstring;
begin
  with wv.ls[w]^ do
  begin
    skiplabel(f); readln(f, par.alert);
    skiplabel(f); readln(f, par.comment);
    skiplabel(f); readln(f, par.dt);
    skiplabel(f); readln(f, par.gen);
    skiplabel(f); readln(f, par.pt);
    skiplabel(f); readln(f, par.pt_gl);
    skiplabel(f); readln(f, par.scan);
    skiplabel(f); readln(f, par.scan_gl);
    skiplabel(f); readln(f, par.sh);
    skiplabel(f); readln(f, par.skip);
    skiplabel(f); readln(f, par.timeperpt);
    skiplabel(f); readln(f, par.vstop);
    skiplabel(f); readln(f, par.yoffset);
    skiplabel(f); readln(f, par.yscale);
    case par.dt of
      dt_COR: with par.cor do
      begin
        skiplabel(f); readln(f, ch);
        skiplabel(f); readln(f, shotsperpt);
        skiplabel(f); readln(f, ts.start);
        skiplabel(f); readln(f, ts.step);
        skiplabel(f); readln(f, ts.stop);
        skiplabel(f); readln(f, ts.t0);
        skiplabel(f); readln(f, ts.wob.ampl);
        skiplabel(f); readln(f, ts.wob.per);
        skiplabel(f); readln(f, ts.wob.ph);
      end;
      dt_ELE: with par.ele do
      begin
        skiplabel(f); readln(f, bs.last);
        skiplabel(f); readln(f, bs.mode);
        skiplabel(f); readln(f, bs.tot);
        skiplabel(f); readln(f, cal.ev);
        skiplabel(f); readln(f, cal.len);
        skiplabel(f); readln(f, cal.t0);
        skiplabel(f); readln(f, dly);
        skiplabel(f); readln(f, reprate);
        skiplabel(f); readln(f, shotsperscan);
        skiplabel(f); readln(f, ts.pos);
        skiplabel(f); readln(f, ts.t0);
        skiplabel(f); readln(f, ts.wob.ampl);
        skiplabel(f); readln(f, ts.wob.per);
        skiplabel(f); readln(f, ts.wob.ph);
      end;
      dt_POW: with par.pow do
      begin
        skiplabel(f); readln(f, ch);
        skiplabel(f); readln(f, cal.int);
        skiplabel(f); readln(f, cal.sl);
      end;
      dt_MAS: with par.mas do
      begin
        skiplabel(f); readln(f, cal.int);
        skiplabel(f); readln(f, cal.sl);
        skiplabel(f); readln(f, ch);
        skiplabel(f); readln(f, delay);
        skiplabel(f); readln(f, inv);
        skiplabel(f); readln(f, scantime);
        skiplabel(f); readln(f, vert);
      end;
    end;
    readln(f); { Read blank line separating header from data. }
  end;
  ReadWaveJUpgrade(w);
end;

procedure ReadWaveJUpgrade(w : integer);
{ Modifies version J wave so is compatible with latest version. }
begin
  with wv.ls[w]^.par do
  begin
    case dt of
    DT_ELE:
      begin
        ele.cal.quad := pardf[dt].ele.cal.quad;
        ele.cal.quadoff := pardf[dt].ele.cal.quad;
      end;
    end;
  end;
end;

procedure ReadWaveK(var f : text; w : integer);
var
  s : bufstring;
begin
  with wv.ls[w]^ do
  begin
    skiplabel(f); readln(f, par.alert);
    skiplabel(f); readln(f, par.comment);
    skiplabel(f); readln(f, par.dt);
    skiplabel(f); readln(f, par.gen);
    skiplabel(f); readln(f, par.pt);
    skiplabel(f); readln(f, par.pt_gl);
    skiplabel(f); readln(f, par.scan);
    skiplabel(f); readln(f, par.scan_gl);
    skiplabel(f); readln(f, par.sh);
    skiplabel(f); readln(f, par.skip);
    skiplabel(f); readln(f, par.timeperpt);
    skiplabel(f); readln(f, par.vstop);
    skiplabel(f); readln(f, par.yoffset);
    skiplabel(f); readln(f, par.yscale);
    case par.dt of
      dt_COR: with par.cor do
```

```
      begin
        skiplabel(f); readln(f, ch);
        skiplabel(f); readln(f, shotsperpt);
        skiplabel(f); readln(f, ts.start);
        skiplabel(f); readln(f, ts.step);
        skiplabel(f); readln(f, ts.stop);
        skiplabel(f); readln(f, ts.t0);
        skiplabel(f); readln(f, ts.wob.ampl);
        skiplabel(f); readln(f, ts.wob.per);
        skiplabel(f); readln(f, ts.wob.ph);
      end;
      dt_ELE: with par.ele do
      begin
        skiplabel(f); readln(f, bs.last);
        skiplabel(f); readln(f, bs.mode);
        skiplabel(f); readln(f, bs.tot);
        skiplabel(f); readln(f, cal.ev);
        skiplabel(f); readln(f, cal.len);
        skiplabel(f); readln(f, cal.t0);
        skiplabel(f); readln(f, cal.quad);
        skiplabel(f); readln(f, cal.quadoff);
        skiplabel(f); readln(f, dly);
        skiplabel(f); readln(f, reprate);
        skiplabel(f); readln(f, shotsperscan);
        skiplabel(f); readln(f, ts.pos);
        skiplabel(f); readln(f, ts.t0);
        skiplabel(f); readln(f, ts.wob.ampl);
        skiplabel(f); readln(f, ts.wob.per);
        skiplabel(f); readln(f, ts.wob.ph);
      end;
      dt_POW: with par.pow do
      begin
        skiplabel(f); readln(f, ch);
        skiplabel(f); readln(f, cal.int);
        skiplabel(f); readln(f, cal.sl);
      end;
      dt_MAS: with par.mas do
      begin
        skiplabel(f); readln(f, cal.int);
        skiplabel(f); readln(f, cal.sl);
        skiplabel(f); readln(f, ch);
        skiplabel(f); readln(f, delay);
        skiplabel(f); readln(f, inv);
        skiplabel(f); readln(f, scantime);
        skiplabel(f); readln(f, vert);
      end;
    end;
    readln(f); { Read blank line separating header from data. }
  end;
end;


procedure rebin;
{ Handle dialog for data rebinning function. }
var
  bins : integer;
  i, j, w : integer;
  s : bufstring;
  temp : real;
begin
  if com.cur = com.num then
  begin
    com_err;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  val(s, bins, j);
  com_wr('Bins ' + makestringint(bins), COLORDEBUG);
  if (bins < 1) or (bins > MAXPOINTS) then
  begin
    com_err;
    exit;
  end;
  wv_sel(1);
  sc_sel_off;
  if com_wr_yn('Rebin') = 0 then
    exit;
  for w := 1 to wv.num do
    with wv.ls[w]^ do
      if sel = 1 then
      begin
        for i := 0 to par.pt div bins - 1 do
        begin
          temp := 0;
          for j := 1 to bins do
            temp := temp + data[i * bins + j];
          for j := 1 to bins do
            data[i * bins + j] := temp / bins;
        end;
        com_wr('Done rebin', COLORDEBUG);
        datasaved := 0;
        parsaved := 0;
        if par.dt = dt_ELE then
          par.ele.bs.mode := 0;
        sc.ls[screen].sel := 1; { tag screen for update. }
      end;
  wv_sel_off;
  UpdateSel;
end;

procedure _rescale;
{ Rescale wave so yscale = 1, yoffset = 0. }
var
  i, j : integer;
begin
  wv_sel(1);
  sc_sel_off;
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if sel = 1 then
      begin
        for j := 1 to par.pt do
          data[j] := PtoV(i, j, XAXISMODE_POINTS, sc.ls[screen].gr.
            YAXISMODE) * sgn(par.yscale);
        par.yscale := 1;
        par.yoffset := 0;
        datasaved := 0;
        parsaved := 0;
        if par.dt = dt_ELE then
          par.ele.bs.mode := 0;
        sc.ls[screen].sel := 1; { tag screen for update. }
      end;
  wv_sel_off;
  UpdateSel;
end;
```

```
function rm_extension(s : bufstring) : bufstring;
{ Returns filename s without its file extension. }
var
  i : integer;
begin
  i := length(s);
  while (s[i] <> '.') and (i >= length(s) - 3) do
    dec(i);
  { Check to ensure file extension exists! }
  if i >= length(s) - 3 then
    rm_extension := copy(s, 1, i - 1);
end;

function rounddown(r : real) : longint;
{ Takes expression to next smallest integer if non-integer, regardless of
  sign. }
begin
  if (r < 0) and (r <> trunc(r)) then
    rounddown := trunc(r) - 1
  else
    rounddown := trunc(r);
end;

procedure roundoff(var r : real; s : real);
{ Rounds off r to nearest multiple of s (can be > or < 1). }
begin
  r := round(r / s) * s;
end;

function roundup(r : real) : longint;
{ Takes expression to next largest integer if non-integer, regardless of
  sign. }
begin
  if (r > 0) and (r <> trunc(r)) then
    roundup := trunc(r) + 1
  else
    roundup := trunc(r);
end;

end.
```

4.5. fpesst.pas

```
unit FpesST;
{ History of modifications (please add to BOTTOM of list!):

  Version 1: Begun 2jun94 BJG.

  Procedures and functions beginning with Y through Z for the program FPES.
  See FPES (at appropriate version) for more specific program modification
  notes.
}

interface

uses
  FpesVar;

procedure SaveEnergy(w : integer);
procedure SaveTime(w : integer);
function SaveWave(w : integer) : boolean;
procedure sc_init(n : integer);
procedure sc_resize(n : integer);
procedure sc_sel;
procedure sc_sel_off;
procedure Scan;
procedure ScanCOR(s : bufstring; t : longint);
procedure ScanELE(s : bufstring; t : longint);
procedure ScanMAS(s : bufstring; t : longint);
procedure ScanPOW(var t : longint);
procedure ScanInit(w, blank : integer);
function scanning(d : integer) : boolean;
procedure ScanStop(w : integer);
procedure ScanStopAll;
function sech_sq(pos, fwhm : real) : real;
procedure SetEnergyConversion;
function sgn(r : real) : real;
procedure ShowParams(par_ptr : par_type_ptr; p, w, poffset : integer);
procedure SkipLabel(var f : text);
procedure SmoothEnergy(w : integer; res : real);
procedure SmoothTime(w : integer; res : real);
procedure SmoothWiener(w : integer; res : real);
procedure StackWaves;
procedure Stage;
function StageDelay(r : real) : longint;
function StageLookup(r : real) : real;
procedure StageMove(r : real);
procedure StageMoveDelay(w : integer; r : real);
procedure StageMoveWait(r : real);
procedure SystemControl;
procedure tekset_rd(ch : integer);
procedure tekset_wr;
function TEK_TIMEPERPT_to_code(timeperpt : real) : integer;
procedure tekwrite(wbuf : bufstring);
procedure tekwrite_ver(wbuf : bufstring);
procedure TextMode;
procedure TidyUp;
procedure Time(var nowtime : longint);
function TimebaseToCode(r : real) : integer;
procedure ToggleAddWavesMode(w : integer);
{procedure ToggleBackgroundSubtractionMode;}
procedure ToggleCrosshairsMode;
procedure ToggleCrosshairsWhich;
procedure ToggleSaveMode;
procedure ToggleXAxisMode(scr : integer; mode : integer);
procedure ToggleYAxisMode(scr : integer);
procedure ToggleYOffsetRescale;
procedure tx_dr(tx : tx_type_p);
procedure tx_scr_up(tx : tx_type_p);
procedure tx_wr(tx : tx_type_p; s : string; col : word; scr : integer);
procedure tx_wr_ch(tx : tx_type_p; c : char);

implementation

uses
  crt, dos, FpesCom, FpesAI, FpesJR, FpesUZ, graph, Keys, TPDecl;

procedure SaveEnergy(w : integer);
{ Write simple energy vs. intensity file of wave w. }
var
  f : text; { File variable. }
  i : integer; { Char counter. }
```

```
  ok : boolean; { Flag for user input. }
  p : integer; { Point number. }
  s, s2 : bufstring; { General string. }
begin
  if wv.num = 0 then
    exit;
  { Determine if filename has same directory as dir, edit wavename so
    that 'U' always appears before rest of filename. }
  with wv.ls[w]^ do
  begin
    str(w, s2);
    s2 := 'Wave ' + s2 + ' ';
    if par.dt <> dt_ELE then
    begin
      com_wr(s2 + 'is not ele datatype; not saved.', COLORHL);
      exit;
    end;
    if get_extension(par.fn) = dt_NAME[dt_ELE] then
      s := rm_extension(par.fn) + '.en'
    else
    begin { Old label method: use U prefix. }
      i := FilenameStart(par.fn);
      if i = 1 then
        s := 'U' + copy(par.fn, i, length(par.fn))
      else
        s := copy(par.fn, 1, i - 1) + 'U' + copy(par.fn, i,
          length(par.fn));
    end;
    if not fullpath(s) then
      s := dir + s;
    ok := TRUE;
    if FileExists(s) then
      if com_wr_ynaesc(s2 + 'file exists. Overwrite') < 1 then
        exit;
    com_wr(s2 + 'saved as ' + s, COLORMESS);
    if FileOpenWrite(f, s) = false then
      exit;
    for p := par.pt downto 1 do
      writeln(f, PtoU(w, p, XAXISMODE_convert), '  ', PtoV(w, p,
        XAXISMODE_convert, YAXISMODE_relative));
    close(f);
  end;
end;

procedure SaveTime(w : integer);
{ Save wave in time space (work for any datatype). }
var
  f : text; { File variable. }
  i : integer; { Char counter. }
  ok : boolean; { Flag for user input. }
  p : integer; { Point number. }
  s, s2 : bufstring; { General string. }
begin
  if wv.num = 0 then
    exit;
  with wv.ls[w]^ do
  begin
    str(w, s2);
    s2 := 'Wave ' + s2 + ' ';
    s := rm_extension(par.fn) + '.tim';
    if not fullpath(s) then
      s := dir + s;
    ok := TRUE;
    if FileExists(s) then
      if com_wr_ynaesc(s2 + 'file exists. Overwrite') < 1 then
        exit;
    com_wr(s2 + 'saved as ' + s, COLORMESS);
    if FileOpenWrite(f, s) = false then
      exit;
    for p := 1 to par.pt do
      writeln(f, PtoU(w, p, XAXISMODE_normal), '  ', PtoV(w, p,
        XAXISMODE_normal, sc.ls[sc.cur].gr.YAXISMODE));
    close(f);
  end;
end;

function SaveWave(w : integer) : boolean;
{ Save wave w, asking user's permission if file already exists. Return
  true if saved; false if not. }
var
  f : text;
  i : integer;
  ok : boolean;
  s, s2 : bufstring;
begin
  if wv.num = 0 then
  begin
    savewave := false;
    exit;
  end;
  with wv.ls[w]^ do
  begin
    str(w, s2);
    s2 := 'Wave ' + s2 + ' ';
    s := par.fn;
    if s = '' then { Flag indicating user has not chosen a name yet. }
    begin
      savewave := false;
      exit;
    end;
    if not fullpath(s) then
      s := dir + s;
    ok := TRUE;
    if FileExists(s) then
      if com_wr_ynaesc(s2 + 'file exists. Overwrite') < 1 then
      begin
        savewave := false;
        exit;
      end;
    if FileOpenWrite(f, s) = false then
      exit;
    com_wr(s2 + 'saved.', COLORMESS);
    parsaved := 1;
    datasaved := 1;
    writeln(f, HEAD_WV);
    writeln(f, PARLABEL[PAR_ALERT], LABEL_END_CHAR, par.alert);
    writeln(f, PARLABEL[PAR_COMMENT], LABEL_END_CHAR, par.comment);
    writeln(f, PARLABEL[PAR_DT], LABEL_END_CHAR, par.dt);
    writeln(f, PARLABEL[PAR_GEN], LABEL_END_CHAR, par.gen);
    writeln(f, PARLABEL[PAR_PT], LABEL_END_CHAR, par.pt);
    writeln(f, PARLABEL[PAR_PT_GL], LABEL_END_CHAR, par.pt_gl);
    writeln(f, PARLABEL[PAR_SCAN], LABEL_END_CHAR, par.scan);
    writeln(f, PARLABEL[PAR_SCAN_GL], LABEL_END_CHAR, par.scan_gl);
    writeln(f, PARLABEL[PAR_SH], LABEL_END_CHAR, par.sh);
    writeln(f, PARLABEL[PAR_SKIP], LABEL_END_CHAR, par.skip);
    writeln(f, PARLABEL[PAR_TIMEPERPT], LABEL_END_CHAR, par.timeperpt);
```

```
    writeln(f, PARLABEL[PAR_VSTOP], LABEL_END_CHAR, par.vstop);
    writeln(f, PARLABEL[PAR_YOFFSET], LABEL_END_CHAR, par.yoffset);
    if globalsavemode = 0 then
      writeln(f, PARLABEL[PAR_YSCALE], LABEL_END_CHAR, par.yscale)
    else
      writeln(f, PARLABEL[PAR_YSCALE], LABEL_END_CHAR, 1);
{     for i := USERMIN to USERMAXdt[par.dt] do
      writeln(f, par.user[i]); }
    case par.dt of
      dt_COR: with par.cor do
      begin
        writeln(f, PARLABELDT[DT_COR][PAR_C_CHANNEL], LABEL_END_CHAR, ch);
        writeln(f, PARLABELDT[DT_COR][PAR_C_SHOTSPERPT], LABEL_END_CHAR,
shotsperpt);
        writeln(f, PARLABELDT[DT_COR][PAR_C_STAGESTART], LABEL_END_CHAR, ts.start);
        writeln(f, PARLABELDT[DT_COR][PAR_C_STAGESTEP], LABEL_END_CHAR, ts.step);
        writeln(f, PARLABELDT[DT_COR][PAR_C_STAGESTOP], LABEL_END_CHAR, ts.stop);
        writeln(f, PARLABELDT[DT_COR][PAR_C_STAGET0], LABEL_END_CHAR, ts.t0);
        writeln(f, PARLABELDT[DT_COR][PAR_C_STAGEwobAMPL], LABEL_END_CHAR,
ts.wob.ampl);
        writeln(f, PARLABELDT[DT_COR][PAR_C_STAGEwobPER], LABEL_END_CHAR,
ts.wob.per);
        writeln(f, PARLABELDT[DT_COR][PAR_C_STAGEwobPH], LABEL_END_CHAR, ts.wob.ph);
      end;
      dt_ELE: with par.ele do
      begin
        writeln(f, PARLABELDT[DT_ELE][PAR_E_BS_LAST], LABEL_END_CHAR, bs.last);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_BS_MODE], LABEL_END_CHAR, bs.mode);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_BS_TOT], LABEL_END_CHAR, bs.tot);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_CALEV], LABEL_END_CHAR, cal.ev);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_CALLENGTH], LABEL_END_CHAR, cal.len);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_CALT0], LABEL_END_CHAR, cal.t0);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_CALQUAD], LABEL_END_CHAR, cal.quad);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_CALQUADOFF], LABEL_END_CHAR,
cal.quadoff);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_TIMEFPES], LABEL_END_CHAR, dly);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_REPRATE], LABEL_END_CHAR, reprate);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_SHOTSPERSCAN], LABEL_END_CHAR,
shotsperscan);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_STAGEPOS], LABEL_END_CHAR, ts.pos);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_STAGET0], LABEL_END_CHAR, ts.t0);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_STAGEwobAMPL], LABEL_END_CHAR,
ts.wob.ampl);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_STAGEwobPER], LABEL_END_CHAR,
ts.wob.per);
        writeln(f, PARLABELDT[DT_ELE][PAR_E_STAGEwobPH], LABEL_END_CHAR, ts.wob.ph);
      end;
      dt_POW: with par.pow do
      begin
        writeln(f, PARLABELDT[DT_POW][PAR_P_CHANNEL], LABEL_END_CHAR, ch);
        writeln(f, PARLABELDT[DT_POW][PAR_P_CALINT], LABEL_END_CHAR, cal.int);
        writeln(f, PARLABELDT[DT_POW][PAR_P_CALSLOPE], LABEL_END_CHAR, cal.sl);
      end;
      dt_MAS: with par.mas do
      begin
        writeln(f, PARLABELDT[DT_MAS][PAR_M_CALINT], LABEL_END_CHAR, cal.int);
        writeln(f, PARLABELDT[DT_MAS][PAR_M_CALSLOPE], LABEL_END_CHAR, cal.sl);
        writeln(f, PARLABELDT[DT_MAS][PAR_M_CHANNEL], LABEL_END_CHAR, ch);
        writeln(f, PARLABELDT[DT_MAS][PAR_M_DELAY], LABEL_END_CHAR, delay);
        writeln(f, PARLABELDT[DT_MAS][PAR_M_INV], LABEL_END_CHAR, inv);
        writeln(f, PARLABELDT[DT_MAS][PAR_M_SCANTIME], LABEL_END_CHAR, scantime);
        writeln(f, PARLABELDT[DT_MAS][PAR_M_VERT], LABEL_END_CHAR, vert);
      end;
    end;
    writeln(f, 'DATA', LABEL_END_CHAR); { Prepare for data. }
    if globalsavemode = 0 then
    begin
      savemode := 0;
      for i:= 1 to par.pt do
        writeln(f, data[i]);
    end
    else
    begin
      savemode := 1;
      for i := 1 to par.pt do
        writeln(f, par.yscale * data[i] + par.yoffset);
    end;
    close(f);
    DrawWaveData; { Display s flag beside wave. }
    savewave := true;
  end;
end;

procedure sc_init(n : integer);
{ Set up new screen with some default parameters. }
var
  s2 : bufstring;
begin
  with sc.ls[n] do
  begin
    mode := sc_mode_GR;
    sel := 0;
    ti.on := 0;
    str(n, s2);
    ti.s := 'Use sc ti to change.';
  end;
  with sc.ls[n].gr do
  begin
    cursoru := 0;
    cursorvisible := 0;
    ullim := 0;
    u2lim := 1;
    v1lim := 0;
    v2lim := 1;
    xaxisdt := dt_MIN;
    xaxismode := XAXISMODE_NORMAL;
    xfullmode := 1;
    xh.mode := 0;
    xh.which := 1;
    xh.u[1] := ullim;
    xh.u[2] := u2lim;
    xh.v[1] := v1lim;
    xh.v[2] := v2lim;
    xon := 0;
    yaxismode := YAXISMODE_ABSOLUTE;
    yfullmode := 1;
    yon := 1;
  end;
end;

procedure sc_resize(n : integer);
{ Resizes screen n to limits given by sc.ls[n].bdy. }
var
  i : integer;
begin
  { Declare title block (if disabled, give no vertical height). }
```

```
with sc.ls[n].ti do
begin
  bdy[1].x := sc.ls[n].gr.bdy[1].x;
  bdy[2].x := sc.ls[n].gr.bdy[2].x;
  bdy[1].y := sc.ls[n].gr.bdy[1].y;
  if on = 1 then
    bdy[2].y := bdy[1].y + textsize
  else
    bdy[2].y := bdy[1].y;
end;
{ Now do graphics-specific areas. }
with sc.ls[n].gr do
begin
{ Declare axis, number and label x & y positions in two stages - first, the
  "short" dimension, whose sizes are well-defined... }
  with ylabel do
  begin
    x1 := bdy[1].x;
    x2 := x1 + yon * textsize;
  end;
  with ynumbers do
  begin
    x1 := ylabel.x2;
    x2 := x1 + yon * MAXYDIGITS * textsize;
  end;
  with yaxis do
  begin
    x1 := ynumbers.x2;
    x2 := x1 + yon * textsize;
  end;
{ Note reverse order of declarations for the following three entries: }
  with xlabel do
  begin
    y2 := bdy[2].y;
    y1 := y2 - xon * textsize;
  end;
  with xnumbers do
  begin
    y2 := xlabel.y1;
    y1 := y2 - xon * textsize;
  end;
  with xaxis do
  begin
    y2 := xnumbers.y1;
    y1 := y2 - xon * textsize;
  end;
{ ...now we go back to fill in the "long" dimensions, which simply take up
  whatever room is left over. }
  with ylabel do
  begin
    y1 := sc.ls[n].ti.bdy[2].y;
    y2 := xaxis.y1 + xon * yon * textsize div 2;
    xmax := (x2 + 1 - x1) div textsize;
    ymax := (y2 + 1 - y1) div textsize;
  end;
  with ynumbers do
  begin
    y1 := ylabel.y1;
    y2 := ylabel.y2;
    xmax := (x2 + 1 - x1) div textsize;
    ymax := (y2 + 1 - y1) div textsize;
    maxynums := ymax div 3; { Calculate maximum
      number of numbers to be printed along y axis; since each number only
      takes up one character height, give space of two characters between
      each for readability (hence factor of 3). }
  end;
  with yaxis do
  begin
    y1 := ylabel.y1 + yon * textsize div 2 + 1 - yon; { Ticks centered on number
text. }
    y2 := ylabel.y2 - yon * textsize div 2 - 1 + yon;
    xmax := (x2 + 1 - x1) div textsize;
    ymax := (y2 + 1 - y1) div textsize;
  end;
  with xlabel do
  begin
    x1 := yaxis.x2 - xon * yon * (MAXXDIGITS div 2) *
      textsize; { Endpoint of y axis, minus half the distance taken up
      by an x number of maximum width. }
    x2 := bdy[2].x;
    xmax := (x2 + 1 - x1) div textsize;
    ymax := (y2 + 1 - y1) div textsize;
  end;
  with xnumbers do
  begin
    x1 := xlabel.x1;
    x2 := xlabel.x2;
    xmax := (x2 + 1 - x1) div textsize;
    ymax := (y2 + 1 - y1) div textsize;
    maxxnums := xmax div (MAXXDIGITS + 1);
    { Calculate maximum number of numbers to be printed along x axis, allow-
      ing space of textsize between each number. }
  end;
  with xaxis do
  begin
    x1 := xlabel.x1 + xon * MAXXDIGITS div 2 * textsize + 1 - xon;
    x2 := xlabel.x2 - xon * MAXXDIGITS div 2 * textsize - 1 + xon;
    xmax := (x2 + 1 - x1) div textsize;
    ymax := (y2 + 1 - y1) div textsize;
  end;
  with plotarea do
  begin
    x1 := xaxis.x1;
    x2 := xaxis.x2;
    y1 := yaxis.y1;
    y2 := yaxis.y2;
    xmax := (x2 + 1 - x1) div textsize;
    ymax := (y2 + 1 - y1) div textsize;
  end;
  { Calculate screen coordinates of xh. }
  for i := 1 to 2 do
  begin
    xh.x[i] := round(UtoX(xh.u[i], n));
    xh.y[i] := round(VtoY(xh.v[i], n));
  end;
 end;
end;

procedure sc_sel;
{ Examines command line for several levels of flags: null, valid wave
  number list, or 'all'. Unlike wv_sel above, 'sel' is not so convenient
  to implement, since it requires cursor moving around on screen titles,
  not all of which may even be visible. So I've settled for this entirely
  adequate compromise.

  Sets wv.sel to 0 if no waves selected (if wv.num = 0, or if sel
```

```
  option returned no tags); otherwise, sets to 1. Leaves com.cur pointing
  to last recognized word (initial command, 'all', or last enumeration
  list number.
}
var
  dummy : integer; { Dummy variable for val. }
  exitflag : boolean; { Flag for enumeration list. }
  i : integer; { wave counter. }
  s : bufstring; { Holds current word. }
begin
  { Set tags to 0. }
  for i := 1 to sc.num do
    sc.ls[i].sel := 0;
  if com.cur = com.num then
  { Null. }
  begin
    sc.ls[sc.cur].sel := 1;
    sc.sel := 1;
    exit;
  end;
  inc(com.cur);
  s := com.ls[com.cur];
  val(s, i, dummy);
  sc.sel := 0;
  if (i >= 1) and (i <= sc.num) then
  begin
    sc.ls[i].sel := 1;
    sc.sel := 1;
    { Read in more wave numbers. }
    exitflag := false;
    while (com.cur < com.num) and (not exitflag) do
    begin
      inc(com.cur);
      val(com.ls[com.cur], i, dummy);
      if (i >= 1) and (i <= sc.num) then
        sc.ls[i].sel := 1
      else
      begin
        dec(com.cur); { Let calling routine figure out what was
          meant. }
        exitflag := true;
      end;
    end;
  end
  else if s = 'all' then
  begin
    for i := 1 to sc.num do
      sc.ls[i].sel := 1;
    sc.sel := 1;
  end
  else
  begin { Unrecognized word; assume want current wave selected. }
    sc.ls[sc.cur].sel := 1;
    sc.sel := 1;
    dec(com.cur); { Leave word for calling routine to decipher. }
  end;
end;

procedure sc_sel_off;
{ Clears all sc.ls[].sel tags. }
var
  i : integer;
begin
  for i := 1 to sc.num do
    sc.ls[i].sel := 0;
  sc.sel := 0;
end;

{*********************************************************************
                          MAIN SCANNING ROUTINE
 *********************************************************************}
procedure Scan;
{ Read A/D card channels, check time elapsed on all waves and read data
  if needed. }
var
  s : bufstring; { 'Wave XXX' storage. }
  t : longint; { Current time. }
  w : integer; { Wave counter. }
begin
  ScanPOW(t); { Read A/D card, update power scans, draw osc. }

  { Find active wave for scanwave to point to. If unsuccessful, scanwave
    = 0. Note that scanwave will continue to point to same active wave
    until wv.ls[]^.scan.done = 1 (because some processes require multiple
    passes through this loop before the scan is done -- i.e. starting
    electron scan, then reading 30 sec later; redoing scan if data
    transfer is bad, etc.) }
  if scanwave = 0 then
    w := 1
  else
    w := scanwave;
  scanwave := 0;
  while w <= wv.num do
    with wv.ls[w]^ do
      if par.dt <> dt_POW then
      begin
        if scan.mode > 1 then
        begin
          scanwave := w;
          w := wv.num + 1; { Trick to get out of loop. }
        end
        else
        begin
          if scan.mode = 1 then { Reset scan.mode to "new" position
            before moving on. }
          begin
            scan.mode := SCAN_MODE_NEW;
            DrawWaveData; { Erase old scanwave indicator. }
          end;
          inc(w); { Go to next wave. }
        end;
      end
      else
        inc(w);

  { Check that scanwave is still valid (otherwise indicates no active
    waves). }
  if scanwave > 0 then
  begin
    str(scanwave, s);
    s := 'Wave ' + s + ' ';
    case wv.ls[scanwave]^.par.dt of
      dt_COR: ScanCOR(s, t);
      dt_ELE: ScanELE(s, t);
      dt_MAS: ScanMAS(s, t);
      { dt_POW handled already -- see above. }
```

```
    end;
  end;
  EraseOsc;
end;

procedure ScanCOR(s : bufstring; t : longint);
{ Scan.mode values for COR scans:
  SCAN_MODE_NEW: Nothing done yet.
  3: Stage still moving.
  2: Taking data.
  1: Ready to move to next wave.
  (0: Not scanning.) }
var
  devname : nbuf; { Device name. }
  j : integer; { General counter. }
  k : integer; { " }
  result : integer; { Boogey variable for val. }
  r : real; { General real. }
  s2 : bufstring; { General strings. }
  w : integer; { Wave counter. }
begin
  with wv.ls[scanwave]^ do
  begin
    { Done anything yet? }
    if scan.mode = SCAN_MODE_NEW then
    begin
      if scan.cycle mod (par.skip + 1) = 0 then
      begin
        { Move stage to beginning, reset counters. }
        DrawWaveData; { Show new scanmode indicator. }
        if par.alert = 1 then
        begin { Pause before beginning. }
          com_wr(s + 'press any key to begin.', COLORHL);
          if readkey = EXTENDED then
            readkey;
        end;
        str(par.scan + 1, s2);
        com_wr(s + 'scan ' + s2 + ' started.', COLORMESS);
        StageMoveDelay(scanwave, par.cor.ts.start);
        scan.mode := 3;
        par.pt := 0;
        AD.ls[par.cor.ch].on := 1;
      end
      else
        scan.mode := 1; { Move on to next wave. }
      inc(scan.cycle);
    end
    else if scan.mode = 3 then
      { See if stage is still moving. }
      if t - scan.starttime < scan.steptime then
        exit
      else
      begin
        scan.shots := 0;
        scan.accum := 0;
        scan.mode := 2;
      end;
    if scan.mode = 2 then
    begin
      { Take all shots at one go, updating power scans as well. }
      if scan.shots < par.cor.shotsperpt then
      begin
        { Add value to accumulator. }
        scan.accum := scan.accum + AD.ls[par.cor.ch].result;
        inc(scan.shots);
        exit;
      end;
      { Add averaged point value to array, update counters. }
      inc(par.pt);
      data[par.pt] := data[par.pt] + scan.accum / scan.shots;
      datasaved := 0; { Set to 0 after 1st successful point. }
{     scan.accum := 0;
      scan.shots := 0; }
      { See if more points. }
      if par.pt < par.pt_gl then
      begin
        { Advance stage. }
        StageMoveDelay(scanwave, par.cor.ts.start + par.pt *
          par.cor.ts.step);
{       com_wr('Press any key to continue', COLORHL);
        if readkey = EXTENDED then
          readkey; }
        scan.mode := 3;
        exit;
      end;
      { Update counters. }
      inc(par.scan);
{     par.pt := 0; }
      { See if more scans. }
      if par.scan < par.scan_gl then
        scan.mode := SCAN_MODE_NEW { Start over at big stage move. }
      else
      begin
        { Send ack code. }
        write(COM2, ACK);
        if auto.off = 1 then
        begin
          { Check all waves for active electron datatypes. If find
            any, keep scan going. Otherwise, turn off. }
          scan.mode := 0;
          for w := 1 to wv.num do
            if (wv.ls[w]^.scan.mode > 0) and (wv.ls[w]^.par.dt =
              dt_ELE) then
              scan.mode := 1;
        end
        else
          scan.mode := 1;
        AutoSave(scanwave);
        ADoff(par.cor.ch);
{       par.scan := 0; { Reset counter for next round. }
      end;
      { Update screen after each scan number. }
      if auto.cor = 1 then
      begin
        FitY(dt_COR); { Includes screen update. }
        do_wv_info(scanwave, 'fwhm');
      end
      else
      begin
        UpdateVitals;
        Update(screen);
      end;
    end;
  end;
end;
```

```
procedure ScanELE(s : bufstring; t : longint);
{ Handle regular and bs ele scans. }
var
  accum : longint;
  j, k : integer;
  result : integer;
  s2 : bufstring;
  temp : real;

  procedure ScanELE_status;
  { Print updated counts for current wave. }
  begin
    with wv.ls[scanwave]^ do
      com_wr('wv ' + makestringint(scanwave) + ' scan ' +
        makestringint(par.scan) + ' tot ' + makestringint(round(
        par.ele.bs.tot)) + ' +' + makestringint(par.ele.bs.
        last), COLORMESS);
  end;

begin
  { BS waves only: When starting a new wave, we first check value of
    _bs.fg:
    0: nothing done, must do bg wave first. Stores scanwave in _bs.fg
      for second pass.
    >0: bg done, now reassign scanwave with _bs.fg and run this wave. }
  if (wv.ls[scanwave]^.par.ele.bs.mode = 1) and (_bs.sts = 0) and
    (_bs.fg = 0) then
  begin
    if scanwave = _bs.bg then
    begin
      wv.ls[scanwave]^.scan.mode := 1; { Skip bg wave. }
      exit;
    end;
    _bs.fg := scanwave; { Store fg wave number for later. }
    scanwave := _bs.bg; { Do bg wave instead. }
    wv.ls[scanwave]^.scan.mode := SCAN_MODE_NEW;
  end;

  if (_bs.sts = 1) and (scanwave = _bs.bg) then
    wv.ls[scanwave]^.scan.mode := 1; { Skip bg wave. }

  str(scanwave, s);
  s := 'Wave ' + s + ' ';
  with wv.ls[scanwave]^ do
  begin
    { Has nothing happened yet? }
    if scan.mode = SCAN_MODE_NEW then
    begin
      { Check cycle count. }
      if scan.cycle mod (par.skip + 1) <> 0 then
        scan.mode := 1 { Skip to next wave. }
      else
      { Move stage to correct location. }
      begin
        DrawWaveData; { Show new scanwave indicator. }
        if par.alert = 1 then
        begin { Pause while waiting to begin. }
          com_wr(s + 'press any key to begin.', COLORHL);
          if readkey = EXTENDED then
            readkey;
        end;
{       str(par.scan + 1, s2);
        com_wr(s + 'scan ' + s2 + ' started.', COLORMESS); }
        StageMoveDelay(scanwave, par.ele.ts.pos);
        scan.mode := 4;
      end;
      inc(scan.cycle);
    end;
    { Is stage moving? }
    if scan.mode = 4 then
    begin
      if t - scan.starttime < scan.steptime then
        exit
      else
        scan.mode := 3;
    end;
    { Stage in position? }
    if scan.mode = 3 then
    begin
      scan.mode := 2;
      { Start the scan -- including TEK if in STS mode. }
      if MCS_init(scanwave) = false then
      begin
        ScanStop(scanwave);
        exit;
      end;
      { Update screen from last pass. }
      UpdateSel;
      exit;
    end;
    { MCS started? }
    if scan.mode = 2 then
    begin
      { Wait some more? }
      if t - scan.starttime < scan.steptime then
        exit;
      { See if MCS is really ready. }
      mwrite(mcs.addr, 'scan?', false);
{      delay(250); }
      mread(mcs.addr, 10);
      Val(rdbuf, scan.shots, result);
      if scan.shots < par.ele.shotsperscan then
        exit;

      { If STS mode, also make sure TEK is ready. }
      if _bs.sts = 1 then
      begin
        { See if TEK is really ready. }
        mwrite(tek, 'acq:state?', true);
        mread(tek, 2);
        if ibbuf[1] <> '0' then
          exit;
        { Read data. }
        tekwrite('curve?');
        if dataread(tek, 2 * wv.ls[_bs.bg]^.par.pt, _bs.bg) = false
          then
        begin
          com_wr('Error reading TEK data. Repeating', COLORHL);
          scan.mode := 3; { Re-issue "go" command for next pass. }
          exit;
        end;
      end;

      { Read MCS data and confirm that read was good. }
{      com_wr(s + 'reading.', COLORMESS); }
      mwrite(mcs.addr, 'binb?', false);
```

```
{     delay(250); }
    if dataread(mcs.addr, 2 * par.pt, scanwave) = false then
    begin
      scan.mode := 3; { Re-issue "go" command for next pass. }
      exit;
    end;

    sc_sel_off; { Blank screen selections for update list. }

    { Add data to array, update counters. }
    datasaved := 0; { Set to 0 after 1st successful scan. }
    for j := 1 to par.pt do
      data[j] := data[j] + tmp[j];
    inc(par.scan);

    if _bs.sts = 1 then
    { STS handling: }
    begin
      { Transfer data to bg wave, scaling data appropriately. }
      with wv.ls[_bs.bg]^ do
      begin
        par.ele.bs.last := 0;
        for j := 1 to _bs.sts_blank - 1 do { Blank out noise spike. }
          tmp[j] := 0;
        for j := _bs.sts_blank2 + 1 to par.pt do
          tmp[j] := 0;
        for j := _bs.sts_blank to _bs.sts_blank2 do
        begin
          temp := (32767 - tmp[j]) * _bs.sts_vert * _bs.sts_fac;
          data[j] := data[j] + temp;
          par.ele.bs.last := par.ele.bs.last + temp;
        end;
        par.ele.bs.tot := par.ele.bs.tot + par.ele.bs.last;
        inc(par.scan);
        sc.ls[screen].sel := 1; { Select screen for update. }
      end;

      { Scale counts by shotsperscan ratios: }
      par.ele.bs.last := wv.ls[_bs.bg]^.par.ele.bs.last *
        par.ele.shotsperscan / wv.ls[_bs.bg]^.par.ele.shotsperscan;
      { Correction for non-toggle mode: record double the number
        of (bg) counts): }
      if par.ele.bs.mode = 0 then
        par.ele.bs.last := par.ele.bs.last * 2;
      par.ele.bs.tot := par.ele.bs.tot + par.ele.bs.last;
      ScanELE_status;
    end
    else if (par.ele.bs.mode = 1) and (_bs.sts = 0) then
    { Non-STS BS handling: }
    begin
      if scanwave <> _bs.fg then
      { We're on bg wave, so calculate total counts. }
      begin
        par.ele.bs.last := 0;
        for j := 1 to par.pt do
          par.ele.bs.last := par.ele.bs.last + tmp[j];
        { Add to total counts of bg wave. }
        par.ele.bs.tot := par.ele.bs.tot + par.ele.bs.last;
        { Now save values in fg wave, set up for fg scan. }
        scanwave := _bs.fg;
        with wv.ls[scanwave]^ do
        begin
          par.ele.bs.last := wv.ls[_bs.bg]^.par.ele.bs.last *
            par.ele.shotsperscan / wv.ls[_bs.bg]^.par.ele.
            shotsperscan; { Scale by ratio of shotsperscan's. }
          par.ele.bs.tot := par.ele.bs.tot + par.ele.bs.last;
          { Print status information - pretend scan # is 1 more. }
          inc(par.scan);
          ScanELE_status;
          dec(par.scan);
        end;
      end
      else
      { fg wave chores: make next scan bg again. }
        _bs.fg := 0;
    end
    else
    begin { Non-bs mode chores : record counts and display. }
      par.ele.bs.last := 0;
      for j := 1 to par.pt do
        par.ele.bs.last := par.ele.bs.last + tmp[j];
      { Add to total counts of bg wave. }
      par.ele.bs.tot := par.ele.bs.tot + par.ele.bs.last;
      ScanELE_status;
    end;

    { Handle mon wave. }
    if mon.w > 0 then
    begin
      mon.tot := 0;
      for j := 1 to par.pt div mon.bins do
        with wv.ls[mon.w]^ do
        begin
          data[j] := 0;
          for k := 1 to mon.bins do
            data[j] := data[j] + wv.ls[scanwave]^.tmp[
              (j - 1) * mon.bins + k];
          mon.tot := mon.tot + data[j];
        end;
      wv.ls[mon.w]^.par.pt := par.pt div mon.bins;
      wv.ls[mon.w]^.par.timeperpt := par.timeperpt * mon.bins;
      with sc.ls[wv.ls[mon.w]^.screen].ti do
        if on = 1 then
          s := 'Tot ' + makestringint(mon.tot);
    end;

    { No more scans? Two ways to terminate:
      1. if par.scan_gl > 0, par.scan must be >= par.scan_gl;
      2. if par.vstop > 0, vmax must be >= par.vstop. }
    UpdateVLimits;
    if ((par.vstop > 0) and (vmax >= par.vstop)) or ((par.scan_gl >
      0) and (par.scan >= par.scan_gl)) then
    begin
      scan.mode := 0;
      AutoSave(scanwave);
      UpdateVitals;
      Update(screen); { Draw final screen. }
    end
    else
      scan.mode := 1;
    { Get screens ready for update on next pass. }
    sc.ls[screen].sel := 1;
    if mon.w > 0 then
      sc.ls[wv.ls[mon.w]^.screen].sel := 1;
  end;
end;
```

```
end;

procedure ScanMAS(s : bufstring; t : longint);
{ Scan.mode values for mass scans:
  SCAN_MODE_NEW: Nothing done yet.
  2: TEK started.
  1: Data read; ready to move to next wave.
 (0: Not scanning.) }
var
  devname : nbuf; { Device name. }
  j : integer; { General counter. }
  k : integer; { " }
  result : integer; { Boogey variable for val. }
  r : real; { General real. }
  s2 : bufstring; { General strings. }
  w : integer; { Wave counter. }
begin
  with wv.ls[scanwave]^ do
       begin
         { Has nothing happened yet? }
         if scan.mode = SCAN_MODE_NEW then
         begin
           if scan.cycle mod (par.skip + 1) = 0 then
           begin
             { Initialize Tek scope. }
             DrawWaveData; { Show new scanmode indicator. }
             if par.alert = 1 then
             begin { Pause while waiting to begin. }
               com_wr(s + 'press any key to begin.', COLORHL);
               if readkey = EXTENDED then
                 readkey;
             end;
             str(par.scan + 1, s2);
             com_wr(s + 'scan ' + s2 + ' started.', COLORMESS);
             { Check that TEK is on GPIB card. }
             devname := 'tek    ';
             tek := ibfind(devname);
             if tek < 0 then
             begin
               com_wr('Tek find error.', COLORHL);
               scan.mode := 0;
               DrawWaveData; { Erase scanmode indicator. }
               exit;
             end;
             ibclr(tek);
             { Shut off possible acquisition in progress (don't use
               tekwrite since this waits for scope to be free first,
               which defeats purpose of shutting off an acquisition!) }
             tekset_rd(wv.ls[scanwave]^.par.mas.ch); { Record settings
               before changing. }
             mwrite(tek, 'acq:state 0', true);
             { No zoom: }
             tekwrite('zoom:state off');
             { Disable fit to screen: }
             tekwrite('hor:fittoscreen off');
             { Average multiple scans: }
             tekwrite('acq:mode ave');
             { Number of scans to average: }
             str(par.scan_gl, s);
             tekwrite('acq:numavg ' + s);
             { Stop after set number of scans: }
             tekwrite('acq:stopafter seq');
             { Set up string for use by channel commands: }
             s := 'ch' + chr(ord('0') + par.mas.ch);
             { Set channel for data acquisition: }
             tekwrite('data:source ' + s);
             { Turn on channel: }
             tekwrite('sel:' + s + ' on');
             { Set vert. position and offset to 0: }
{             tekwrite(s + ':offset 0'); }
             tekwrite(s + ':pos 0');
             { Read vert. scale, place in par.mas.vert. }
             tekwrite(s + ':scale?');
             mread(tek, sizeof(s));
             s := '';
             for j := 1 to ibcnt - 1 do
               s := s + ibbuf[j];
             val(s, par.mas.vert, j); { j is dummy variable. }
             { 2 bytes per bin: }
             tekwrite('data:width 2');
             { Transfer desired number of data points: }
             str(par.pt, s);
             tekwrite('data:start 1;stop ' + s);
             { Use signed binary format with LSB (low byte) transferred
               first: }
             tekwrite('data:encdg sribinary');
             { Calculate minimum # of bins and set: }
             j := round(par.pt);
             if j <= 500 then
               j := 500
             else if j <= 1000 then
               j := 1000
             else if j <= 2500 then
               j := 2500
             else
               j := 5000;
             str(j, s);
             tekwrite('hor:reco ' + s);
             { Set pretrigger to 0%: }
             tekwrite('hor:trig:pos 0');
             { Check delay parameter to determine whether to use main
               or delayed trigger: }
             if par.mas.delay = 0 then
             begin
               { Use main sweep: }
               tekwrite('hor:mode main');
               { Set time per division ( ='time per pt * pts per div): }
               str(par.timeperpt * TEK_PTPERDIV, s);
               tekwrite('hor:main:scale ' + s);
             end
             else
             begin
               { Use delayed sweep: }
               tekwrite('hor:mode delayed');
               { Set time per division ( = time per pt * pts per div): }
               str(par.timeperpt * TEK_PTPERDIV, s);
               tekwrite('hor:delay:scale ' + s);
               { Set delay time before starting acquisition: }
               str(par.mas.delay, s);
               tekwrite('hor:delay:time:runsafter ' + s);
             end;
             { Start the scan: }
             tekwrite('acq:state run');

             Time(scan.starttime); { Record start time. }
             scan.steptime := 100 * round(par.scan_gl) div TEKREPRATE;
```

```
            { How long to wait between data reads. }
            scan.mode := 2;
          end
          else
            scan.mode := 1; { Skip to next wave. }
          inc(scan.cycle);
        end;
        { TEK started? }
        if scan.mode = 2 then
        begin
          { Wait some more? }
          if t - scan.starttime < scan.steptime then
            exit;
          { See if TEK is really ready. }
          mwrite(tek, 'acq:state?', true);
          mread(tek, 2);
          if ibbuf[1] <> '0' then
            exit;
          { Read data. }
          tekwrite('curve?');
          if dataread(tek, 2 * par.pt, scanwave) = false then
          begin
            com_wr('Error reading TEK data.', COLORHL);
            scan.mode := 0;
            exit;
          end;
          ibclr(tek); { Reset tek for user. }
          tekset_wr; { Restore settings. }
          { Add data to array, update counters. }
          datasaved := 0; { Set to 0 after 1st successful scan. }
          { Transfer data to wave, checking invert flag first: }
          if par.mas.inv = 0 then
            for j := 1 to par.pt do
              data[j] := tmp[j]
          else
            for j := 1 to par.pt do
              data[j] := -tmp[j];
          scan.mode := 0;
          UpdateVitals;
          Update(screen);
        end;
      end;
end;

procedure ScanPOW(var t : longint);
{ Read A/D card, update power waves, handle time wraparound, draw osc
  if on. }
var
  ptadv : integer; { Number of points to advance in power scans. }
  s : bufstring;
  w : integer;
begin
  ADReadAll; { Read active channels on A/D card. }
  time(t); { Read current time. }
  DrawOsc;

  { Handle time wraparound: Normally t is always larger than starttime,
    but if clock has just reset (at 12 AM), will be smaller. In this case
    we just calculate the amount of time after 12 AM at which we need to
    wait, and adjust starttimes accordingly. }
  for w := 1 to wv.num do
    with wv.ls[w]^ do
      if (scan.mode > 0) and (t < scan.starttime) then
        dec(scan.starttime, TIMEMAX - scan.steptime);

  { Handle power scans. }
  for w := 1 to wv.num do
    with wv.ls[w]^ do
      if (scan.mode > 0) and (par.dt = dt_POW) then
      begin
        { Accumulate signal, update shots. }
        scan.accum := scan.accum + AD.ls[par.pow.ch].result;
        inc(scan.shots);
        { Time to advance point counter? }
        ptadv := (t - scan.starttime) div scan.steptime;
        if ptadv > 0 then
        begin
          { Save averaged power in data array, update counters. }
          inc(par.pt);
          data[par.pt] := scan.accum / scan.shots;
          if ptadv > 1 then
            inc(par.pt, ptadv - 1); { Skip over intervening points which
              may have been lost (due to user tying up computer). }
          datasaved := 0; { Set to 0 after 1st successful point. }
          scan.accum := 0;
          scan.shots := 0;
          Time(scan.starttime); { Start next point. }
          { See if enough points. }
          if par.pt >= par.pt_gl then
          begin
            { Indicate scan is finished, update counters, screen. }
            scan.mode := 0;
            par.pt := par.pt_gl; { Ensure points is not larger than
              max. }
            ADoff(par.pow.ch); { Turn off channel. }
            str(w, s);
            com_wr('Wave ' + s + ' scan completed.', COLORMESS);
            UpdateVitals;
            Update(screen);
            AutoSave(w);
          end;
        end;
      end;
end;

procedure ScanInit(w, blank : integer);
{ Sets needed variables when scan is started. blank is flag to reset
  scan number and blank data. }
begin
  with wv.ls[w]^ do
  begin
    if blank = 1 then
    begin
      datasaved := 1; { Initially just blank data so don't care if
        not saved. }
      FillChar(data, sizeof(data), 0);
      FillChar(tmp, sizeof(tmp), 0);
      par.scan := 0;
      { Initialize bg total counts. }
      if par.dt = dt_ELE then
      begin
        par.ele.bs.tot := 0;
        par.ele.bs.last := 0;
      end;
    end;
    scan.cycle := 0;
```

```
    scan.mode := SCAN_MODE_NEW;
    case par.dt of
      DT_COR:
      begin
        par.pt := 0;
        par.cor.ts.wob := ts.wob;
      end;
      DT_ELE:
        par.ele.ts.wob := ts.wob;
      DT_POW:
      begin
        par.pt := 0;
        ad.ls[par.pow.ch].on := 1; { Activate A/D channel. }
        scan.steptime := round(100 * par.timeperpt);
        Time(scan.starttime); { Start running immediately. }
        com_wr('Wave ' + makestringint(w) + ' scanning.', COLORMESS);
      end;
    end;
  end;
end;

function scanning(d : integer) : boolean;
{ Determines if any d dt waves are active. }
var
  w : integer;
begin
  for w := 1 to wv.num do
    if (wv.ls[w]^.par.dt = d) and (wv.ls[w]^.scan.mode > 0) then
    begin
      scanning := true;
      exit;
    end;
  scanning := false;
end;

procedure ScanStop(w : integer);
{ Terminate scan in w, if one exists. }
var
  s : bufstring; { General string. }
begin
  if wv.num = 0 then
    exit;
  with wv.ls[w]^ do
    if scan.mode > 0 then
    begin
      scan.mode := 0;
      com_wr_wv(w, 'done.', COLORMESS);
      case par.dt of
      dt_COR:
        { Data will be a mess if took more than one scan, because data will
          only be partially overwritten. User will decide if useful, but as
          far as number of scans goes, I will report number of COMPLETE
          scans. If only one scan taken, reduces number of points to #
          actually taken. }
        begin
{         if par.scan > 0 then
            par.scan_gl := par.scan
          else if par.pt > 0 then
          begin
            par.scan_gl := 1;
            par.pt_gl := par.pt;
          end; }
          ADoff(par.cor.ch); { Turn off channel if no others using it. }
{         Update(screen); }
        end;
{     dt_ELE:
        Update(screen); }
      dt_POW:
        begin
          if scan.shots > 0 then { Evaluate last point. }
          begin
            inc(par.pt);
            data[par.pt] := scan.accum / scan.shots;
          end;
{         if par.pt > 0 then
            par.pt_gl := par.pt; }
          ADoff(par.pow.ch); { Turn off channel if no others using it. }
{         Update(screen); }
        end;
      end;
    end;
end;

procedure ScanStopAll;
{ Stop all waves. }
var
  i : integer;
begin
  sc_sel_off;
  for i := 1 to wv.num do
    ScanStop(i);
  UpdateSel;
end;

function sech_sq(pos, fwhm : real) : real;
{ Returns sech^2(pos / fwhm). }
const
  CONV = 1.762747174;
var
  x : real;
begin
  if (fwhm = 0) then
  begin
    sech_sq := 0;
    exit;
  end;
  x := CONV * pos / fwhm;
  sech_sq := 4 / sqr(exp(x) + exp(-x));
end;

procedure SetEnergyConversion;
{ Allows user to input energy conversion factors (PAR_E_CALLENGTH,
  PAR_E_CALT0) which are applied to all existing ELECTRON waves. }
var
  p : real; { General power. }
  s : bufstring; { General string. }
  w : integer; { Wave counter. }
begin
  textmode;
  clrscr;
  p := Power(setenergyconversionlength);
  s := 'Enter calibration length (default = ' + makestring(
    setenergyconversionlength / p, MAXXDIGITS, MAXXDIGITS - 1) + ' ' +
    UnitPrefix(round(log10(p))) + 'm):';
  writeln(s);
  if macro_override = 0 then
```

```
begin
  readln(s);
  if s <> '' then
  begin
    val_r(s, setenergyconversionlength);
    setenergyconversionlength := setenergyconversionlength * p;
  end;
end;
p := Power(setenergyconversiont0);
s := 'Enter calibration T0 (default = ' + makestring(
  setenergyconversiont0 / p, MAXXDIGITS, MAXXDIGITS - 1) + ' ' +
  UnitPrefix(round(log10(p))) + 's):';
writeln(s);
if macro_override = 0 then
begin
  readln(s);
  if s <> '' then
  begin
    val_r(s, setenergyconversiont0);
    setenergyconversiont0 := setenergyconversiont0 * p;
  end;
end;
p := Power(setenergyconversionquad);
s := 'Enter calibration quad factor (default = ' + makestring(
  setenergyconversionquad / p, MAXXDIGITS,
  MAXXDIGITS - 1) + ' ' + UnitPrefix(round(log10(p))) + 'eV):';
writeln(s);
if macro_override = 0 then
begin
  readln(s);
  if s<> '' then
  begin
    val_r(s, setenergyconversionquad);
    setenergyconversionquad := setenergyconversionquad * p;
  end;
end;
p := Power(setenergyconversionquadoff);
s := 'Enter calibration quad factor (default = ' + makestring(
  setenergyconversionquadoff / p, MAXXDIGITS,
  MAXXDIGITS - 1) + ' ' + UnitPrefix(round(log10(p))) + 'eV):';
writeln(s);
if macro_override = 0 then
begin
  readln(s);
  if s<> '' then
  begin
    val_r(s, setenergyconversionquadoff);
    setenergyconversionquadoff :=
      setenergyconversionquadoff * p;
  end;
end;
p := Power(setenergyconversionlaserev);
s := 'Enter calibration laser energy (default = ' + makestring(
  setenergyconversionlaserev / p, MAXXDIGITS, MAXXDIGITS - 1) + ' ' +
  UnitPrefix(round(log10(p))) + 'eV):';
writeln(s);
if macro_override = 0 then
begin
  readln(s);
  if s <> '' then
  begin
    val_r(s, setenergyconversionlaserev);
    setenergyconversionlaserev := setenergyconversionlaserev * p;
    end;
  end;
  for w := 1 to wv.num do
    with wv.ls[w]^.par do
      if dt = dt_ELE then
      begin
        ele.cal.len := setenergyconversionlength;
        ele.cal.t0 := setenergyconversiont0;
        ele.cal.quad := setenergyconversionquad;
        ele.cal.quadoff := setenergyconversionquadoff;
        ele.cal.ev := setenergyconversionlaserev;
      end;
  UpdateAll;
end;

function sgn(r : real) : real;
{ Returns sign of r. }
begin
  if r < 0 then
    sgn := -1
  else
    sgn := 1;
end;

procedure ShowParams(par_ptr : par_type_ptr; p, w, poffset : integer);
{ Write all parameters to screen. w is wave number (0 if editing
  defaults). }
var
  i : integer; { Parameter. }
  r : real; { General real. }
  y : integer;
begin
  textmode;
  clrscr;
  { Print header message. }
  textcolor(LIGHTGREEN);
  if w = 0 then
    writeln_ctr('Defaults', 40)
  else if wv.ls[w]^.scan.mode > 0 then
    writeln_ctr('Wave ' + makestringint(w) + ' active: READ ONLY', 40)
  else
    writeln_ctr('Wave ' + makestringint(w), 40);
  with par_ptr^ do
  if (dt >= DT_MIN) and (dt <= DT_MAX) then
  begin
    y := USERMAXDT[dt] - poffset + PARYSTART - 1;
    if y > PARYSTOP then
      y := PARYSTOP;
    for y := PARYSTART to y do
    begin
      i := poffset + y - PARYSTART + 1;
      gotoxy(CHANGEPARXNAME, y);
      if i < USERMIN then
        textcolor(LIGHTGREEN)
      else
        textcolor(WHITE);
      { Name: }
      if i < USERMIN then
        write(PARLABEL[i])
      else
        write(PARLABELDT[dt][i]);

      { Values: }
```

```
gotoxy(CHANGEPARXVALUE, y);
if i = p then
  textcolor(LIGHTRED)
else if i < USERMIN then
  textcolor(LIGHTGREEN)
else
  textcolor(WHITE);
case i of
PAR_ALERT: write(alert);
PAR_COMMENT: write(comment);
PAR_dt: write(dt_NAME[dt]);
PAR_FILENAME: write(fn);
PAR_GEN: write(gen);
PAR_PT: write(pt);
PAR_PT_GL: write(pt_gl);
PAR_SCAN: write(scan);
PAR_SCAN_GL: write(scan_gl);
PAR_SH: write(wv_sh_NAME[sh]);
PAR_SKIP: write(skip);
PAR_TIMEPERPT:
  if dt <> dt_COR then
    writevalue(timeperpt, 0, VALDEC, VALMAX, 's')
  else
    writevalue(timeperpt, POWFS, VALDECFS, VALMAXFS, 's');
PAR_VSTOP: writevalueunitless(vstop, 0, VALDEC, VALMAX);
PAR_YOFFSET: writevalueunitless(yoffset, 0, VALDEC, VALMAX);
PAR_YSCALE: writevalueunitless(yscale, 0, VALDEC, VALMAX);
else
  case dt of
  dt_COR:
    case i of
    PAR_C_STAGESTART: writevalue(cor.ts.start, POWTS, VALDECTS,
      VALMAXTS, 'm');
    PAR_C_STAGESTEP: writevalue(cor.ts.step, POWTS, VALDECTS,
      VALMAXTS, 'm');
    PAR_C_STAGESTOP: writevalue(cor.ts.stop, POWTS, VALDECTS,
      VALMAXTS, 'm');
    PAR_C_STAGET0: writevalue(cor.ts.t0, POWTS, VALDECTS, VALMAXTS,
      'm');
    PAR_C_STAGEwobAMPL: writevalue(cor.ts.wob.ampl, POWTS,
      VALDECTS, VALMAXTS, 'm');
    PAR_C_STAGEwobPER: writevalue(cor.ts.wob.per, POWTS,
      VALDECTS, VALMAXTS, 'm');
    PAR_C_STAGEwobPH: writevalueunitless(cor.ts.wob.ph, 0,
      VALDEC, VALMAX);
    PAR_C_CHANNEL: write(cor.ch);
    PAR_C_SHOTSPERPT: write(cor.shotsperpt);
    end;
  dt_ELE:
    case i of
    PAR_E_BS_LAST: writevalueunitless(ele.bs.last, 0, VALDEC,
      VALMAX);
    PAR_E_BS_MODE: write(ele.bs.mode);
    PAR_E_BS_TOT: writevalueunitless(ele.bs.tot, 0, VALDEC,
      VALMAX);
    PAR_E_CALEV: writevalue(ele.cal.ev, 0, VALDEC, VALMAX, 'eV');
    PAR_E_CALLENGTH: writevalue(ele.cal.len, 0, VALDEC, VALMAX,
      'm');
    PAR_E_CALT0: writevalue(ele.cal.t0, 0, VALDEC, VALMAX, 's');
    PAR_E_CALQUAD: writevalue(ele.cal.quad, 0, VALDEC, VALMAX, 'm2/s2');
    PAR_E_CALQUADOFF: writevalue(ele.cal.quadoff, 0, VALDEC, VALMAX, 's/m');
    PAR_E_REPRATE: writevalue(ele.reprate, 0, VALDEC, VALMAX, 'Hz');
    PAR_E_STAGEPOS: writevalue(ele.ts.pos, POWTS, VALDECTS,
      VALMAXTS, 'm');
    PAR_E_STAGET0: writevalue(ele.ts.t0, POWTS, VALDECTS, VALMAXTS,
      'm');
    PAR_E_STAGEwobAMPL: writevalue(ele.ts.wob.ampl, POWTS,
      VALDECTS, VALMAXTS, 'm');
    PAR_E_STAGEwobPER: writevalue(ele.ts.wob.per, POWTS,
      VALDECTS, VALMAXTS, 'm');
    PAR_E_STAGEwobPH: writevalueunitless(ele.ts.wob.ph, 0,
      VALDEC, VALMAX);
    PAR_E_TIMEFPES: writevalue(ele.dly, POWFS, VALDECFS, VALMAXFS,
      's');
    PAR_E_SHOTSPERSCAN: write(ele.shotsperscan);
    end;
  dt_POW:
    case i of
    PAR_P_CHANNEL: write(pow.ch);
    PAR_P_CALINT: writevalue(pow.cal.int, 0, VALDEC, VALMAX, 'W');
    PAR_P_CALSLOPE: writevalue(pow.cal.sl, 0, VALDEC, VALMAX, 'W');
    end;
  dt_MAS:
    case i of
    PAR_M_CALINT: writevalue(mas.cal.int, 0, VALDEC, VALMAX, 's');
    PAR_M_CALSLOPE: writevalue(mas.cal.sl, 0, VALDEC, VALMAX,
      'Da/s^2');
    PAR_M_CHANNEL: write(mas.ch);
    PAR_M_DELAY: writevalue(mas.delay, 0, VALDEC, VALMAX, 's');
    PAR_M_INV: write(mas.inv);
    PAR_M_SCANTIME: writevalue(mas.scantime, 0, VALDEC, VALMAX, 's'
      );
    PAR_M_VERT: writevalue(mas.vert, 0, VALDEC, VALMAX, 'V/div');
    end;
  end;
 end;
end;
{ In order to write text in white, must print something to screen -
  so we put a space in the corner. }
gotoxy(79, 25);
textcolor(WHITE);
write(' ');
end;

procedure SkipLabel(var f : text);
{ Reads past label (terminated by LABEL_END_CHAR) in wv or ws file. }
var
  c : char;
begin
  repeat
    read(f, c);
  until c = LABEL_END_CHAR;
end;

procedure SmoothEnergy(w : integer; res : real);
{ This is taken partly from the procedure SMOOTH in the TENURE
  computer code implemented by S. Bradforth.
  smoothed data [E0] = sum over i (weight[Ei-E0] * data[E0])
                       ------------------------------------
                       sum over i  weight[Ei-E0]
}
var
  a : real;
  energyj : real;
```

```
  factor : real;
  i,j : integer;
  max : integer;
  maxdE : real;
  min : integer;
  range : integer;
  realdE : real;
{  root : real; }
  t : real;
  totalwave : real;
  totalweight : real;
  wgt : real;
begin
  a := 4 * LN_2 / res / res; { convert fwhm to std. dev. }
  maxdE := SQRTLN_100 / sqrt(a);
  { set up limit such that smooth iff abs (E0 - Ei) < maxdE so that
    weight (E0-Ei) >= 0.01 }
  with wv.ls[w]^ do
  begin
    factor := 2 * SLOPEFACTOR * par.timeperpt * sqr(par.ele.cal.len);
    for j := pmin to par.pt do begin
      t := j * par.timeperpt - par.ele.cal.t0;
{      root := sqrt(slope*(slope+4*t*quad)); }
      realdE := factor / (t * t * t);
{      realdE := timebase*((slope+root)/t/t/t+quad/t/t*(1-slope/root));
      { realdE is the differential step in energy.  Note that
          realdE/timebase is derivative of E as a function of t }
      range := trunc(maxdE / realdE);
      if range > 1 then begin
        min := j - range;
        if min < pmin then
          min := pmin;
        max := j + range;
        if max > par.pt then
          max := par.pt;
        totalwave := 0.0;
        totalweight := 0.0;
        energyj := PtoU(w, j, xaxismode_CONVERT);
        for i := min to max do begin
          wgt := weight(PtoU(w, i, xaxismode_CONVERT) - energyj, a);
          totalwave := totalwave + wgt * data[i];
          totalweight := totalweight + wgt;
        end;
        wv.temp[j] := totalwave / totalweight;
      end
      else
        wv.temp[j] := data[j];
    end; { loop }
    { Copy data back, blanking out parts below pmin. }
    for j := 1 to pmin - 1 do
      data[j] := 0;
    for j := pmin to par.pt do
      data[j] := wv.temp[j];
  end;
end;

procedure SmoothTime(w : integer; res : real);
{ Smooth wave w in time. }
var
  a : real;
  i, j : integer;
  max, min : integer;
  range : integer;
  s : bufstring;
  timej : real;
  t_wt : real; { Total weight. }
  t_wt_y : real; { Total weight * data. }
  wt : real; { Weight. }
begin
  with wv.ls[w]^ do
  begin
    res := res * 1e-9; { ns -> s. }
    a := 4 * LN_2 / res / res; { convert fwhm to std. dev. }
    range := trunc(SQRTLN_100 / sqrt(a) / par.timeperpt);
    { set up limit such that smooth iff weight (ti - tj) >= 0.01 }
    for j := 1 to par.pt do
    begin
      timej := PtoU(w, j, XAXISMODE_NORMAL);
      if ( range > 1 ) then { smooth this point ! }
      begin
        min := j - range;
        if ( min < 1) then
          min := 1;
        max := j + range;
        if ( max > wv.ls[w]^.par.pt ) then
          max := wv.ls[w]^.par.pt;
        t_wt := 0.0;
        t_wt_y := 0.0;
        for i := min to max do
        begin
          wt := weight(PtoU(w, i, XAXISMODE_NORMAL) - timej, a);
          t_wt := t_wt + wt;
          t_wt_y := t_wt_y + wt * data[i];
        end;
        wv.temp[j] := t_wt_y / t_wt;
      end
      else
        wv.temp[j] := data[j];
    end;
    for j := 1 to par.pt do
      data[j] := wv.temp[j];
  end;
end;

procedure SmoothWiener(w : integer; res : real);
{ Performs time-domain smoothing using a Wiener filter method. This
  approach (taken from TENURE program in S. E. Bradforth's thesis) con-
  sists of a Fourier transform of the energy data, multiplication of the
  transformed data by a Lorentzian filter function, and inverse-Fourier
  transform back to energy space.

  Assumes filter array already set up, since this takes some time to
  generate and can be used for multiple jobs. }
var
  a : fft_array_type; { Storage of transformed data. }
  i, ii : integer;
begin
  with wv.ls[w]^ do begin
    { Quick check that points are compatible. Eventually may want to
      use a padding method so any length wave can be filtered. }
    if par.pt <> MAXPOINTS then
    begin
      com_wr('Wrong number of points', COLORHL);
      exit;
    end;
    { Set up a. }
```

```
    for i := 1 to MAXPOINTS do begin
      a[2 * i - 1] := data[i];
      a[2 * i] := 0;
    end;
    { Fourier transform. }
    fft(a, MAXPOINTS, 1);
    { Filter. }
    for ii := 1 to MAXPOINTS div 2 + 1 do begin
      i := 2 * ii;
      a[i - 1] := a[i - 1] * filter[ii];
      a[i] := a[i] * filter[ii];
    end;
    for ii := MAXPOINTS div 2 + 2 to MAXPOINTS do begin
      i := 2 * ii;
      a[i - 1] := a[i - 1] * filter[MAXPOINTS - ii + 2];
      a[i] := a[i] * filter[MAXPOINTS - ii + 2];
    end;
    { Inverse Fourier transform. }
    fft(a, MAXPOINTS, -1);
    { Transfer back to wave data. }
    for i := 1 to par.pt do
      data[i] := a[i * 2 - 1] / MAXPOINTS;
  end;
end;

{procedure SmoothWaves;
var
  exitflag : boolean;
  k : integer;
  ok : boolean;
  resolution : real;
  w : integer;
begin
  if (scan_.mode = 1) and (scan_.w = wv.cur) then
    exit;
  if (scantype = ENERGYSCAN) then
    begin
      TextMode;
      writeln(' ');
      writeln('Gaussian smoothing of data.');
      exitflag := FALSE;
      repeat
        ok := TRUE;
        writeln(' ');
        writeln('Which wave would you like to smooth? (0 will smooth all waves)');
        writeln('Type a negative value to exit.');
        readln(w);
        if (w > wv.num) then
          begin
            writeln(' ');
            writeln('Inputted wave number out of range.');
            ok := FALSE;
          end;
        if (w < 0) then
          exitflag := TRUE;
      until (ok = TRUE) or (exitflag = TRUE);
      if (exitflag = FALSE) then
        begin
          if (w = 0) then
            writeln('Wavename: "all waves"')
          else
            writeln('Wavename: "'+wave[w]^.wavename+'".');
          repeat
            ok := TRUE;
            writeln(' ');
            writeln('Type full width at half maximum (in meV) of Gaussian');
            writeln('to be used for smoothing.');
            readln(resolution);
            if (resolution <= 0) then
              begin
                writeln('Resolution must be a positive number.');
                ok := FALSE;
              end;
          until (ok = TRUE);
          resolution := resolution / 1000.0; { convert from meV to eV }
{           if (w = 0) then
            for k := 1 to wv.num do
              begin
                SmoothWave(k,resolution);
                wv.ls[k]^.saved := 0;
              end
          else
            begin
              SmoothWave(w,resolution);
              wv.ls[w]^.saved := 0;
            end;
        end; { if }
{       DrawAll;
    end
  else
    begin
      write(#7);
      message := 'Must be an Energy Scan.';
      Drawmessagebox;
    end;
end; { procedure SmoothWaves }

procedure StackWaves;
{ Stack waves vertically. }
var
  a : real;                          { Generic real. }
  c : char;                          { Generic char. }
  s : bufstring;                     { Generic string. }
  w : integer;                       { Wave index. }
begin
  if wv.num = 0 then
    exit;
  a := 0;
  writeln;
  writeln('Enter Y spacing between each wave. A positive value indicates wave 1 is
on top;');
  writeln('a negative value indicates wave 1 is on bottom. Enter 0 to superimpose
all');
  writeln('waves. Previous offset values will be undone. ');
  writeln('Default = ' + makestring(a, MAXXDIGITS, MAXXDIGITS - 1) + ' ('
    + sc.ls[sc.cur].gr.yunits + '): ');
  readln(s);
  if s <> '' then
  begin
    val_r(s, a);
    a := a * sc.ls[sc.cur].gr.ypower;
  end;
  if a >= 0 then { Stack downward from 1st wave. }
    for w := 1 to wv.num do
      wv.ls[w]^.par.yoffset := a * (wv.num - w) / wv.ls[w]^.par.yscale
  else { Stack upward from 1st wave. }
```

```
    for w := 1 to wv.num do
      wv.ls[w]^.par.yoffset := -a * (w - 1) / wv.ls[w]^.par.yscale;
  UpdateAll;
end;

procedure Stage;
{ Allows user to communicate with Aerotech translation stage. }
const
  LF = #10; { Compiler seems to need this declaration here rather than in the
    Keys unit. }
var
  c : char;
  command : string[80];
  exitflag : boolean;
  l : real;
  p : real; { Power of 1000. }
  readAD : boolean;
  sum : real;
  x : integer;
begin
  readAD := false;
  exitflag := FALSE;
  textmode;
  clrscr;
  { Main loop. }
  repeat
    write('Position = ');
    writevalue(ts.pos, POWTS, VALDECTS, VALMAXTS, 'm');
    write('. Step size = ');
    writevalue(move.step, POWTS, VALDECTS, VALMAXTS, 'm');
    writeln('. Type H for help.');
    if readAD = true then
      while not keypressed do
      begin
        writeln(ADRead(1));
        delay(100);
      end;
    case readkey of
    EXTENDED:
      case readkey of
      XARROWUP:
        begin
          move.step := move.step * 2;
          if move.step > STAGEMAX then
            move.step := STAGEMAX;
        end;
      XARROWDOWN: { Down arrow }
        if move.step > ts.step then
          move.step := int((move.step / ts.step)) * ts.step / 2;
      XARROWLEFT: { Left arrow }
        if ts.pos - move.step >= STAGEMIN then
          StageMove(ts.pos - move.step);
      XARROWRIGHT: { Right arrow }
        if ts.pos + move.step <= STAGEMAX then
          StageMove(ts.pos + move.step);
      XHOME:
        begin
          write(COM2, ACK);
          delay(ACKDELAY);
          write(COM2, '#CAHM' + LF);
          writeln('Homing stage. Press any key to send ABSL command.');
          readln;
          write(COM2, ACK);
          delay(ACKDELAY);
          write(COM2, '#CAABSL' + LF);
          ts.pos := 0;
        end;
      end; { Extended keys. }
    'A', 'a':
      begin
        write(COM2, ACK);
        delay(ACKDELAY);
      end;
    'C', 'c':
      begin
        writeln('Enter command to send (begin with # if not CA-type command):');
        readln(command);
        write(COM2, ACK);
        delay(ACKDELAY);
        if command[1] = '#' then
          write(COM2, command + LF)
        else
          write(COM2, '#CA' + command + LF);
      end;
    'H', 'h':
      begin
        clrscr;
        writeln('Up arrow: Double step size.');
        WRITELN('Down arrow: Halve step size.');
        WRITELN('Left arrow: Move stage backward.');
        WRITELN('Right arrow: Move stage forward.');
        WRITELN('ESC: Exit.');
        writeln('HOME: Home stage and place in absolute mode.');
        writeln('A: Send ACK code.');
        writeln('C: Send command.');
        WRITELN('H: Display this help screen.');
        WRITELN('I: Initialize.');
        WRITELN('P: Change position.');
        writeln('R: Read/don"t read A/D converter.');
        WRITELN('S: Change step size.');
        writeln('T: Toggle stage.');
        WRITELN('X: Send disengage command (#EA).');
        WRITELN;
      end;
    'I', 'i':
      begin
        { Send setup commands. }
        writeln('Set up. Please wait.');
        write(COM2, #3);
        DELAY(1000);
        write(COM2, #32);
        DELAY(1000);
        write(COM2, #30);
        DELAY(1000);
        write(COM2, ACK);
        DELAY(ACKDELAY);
        { Home stage. }
        writeln('Homing stage. Press RETURN when ready.');
        write(COM2, '#CAHM' + LF);
        readln;
        write(COM2, ACK);
        { Set absolute mode. }
        write(COM2, '#CAABSL' + LF);
        delay(1000);
        write(COM2, ACK);
        ts.pos := 0;
```

```
  end;
'P', 'p': { Change position }
  begin
    write('Enter position (default = ');
    writevalue(ts.pos, POWTS, VALDECTS, VALMAXTS, 'm');
    writeln('):');
    readvalue(ts.pos, POWTS);
    roundoff(ts.pos, ts.step);
    limit(ts.pos, STAGEMIN, STAGEMAX);
    StageMove(ts.pos);
  end;
'R', 'r': { Toggle read/don't read A/D converter. }
  readAD := not readAD;
'S', 's': { Change step size }
  begin
    write('Enter step size (default = ');
    writevalue(move.step, POWTS, VALDECTS, VALMAXTS, 'm');
    writeln('):');
    readvalue(move.step, POWTS);
    roundoff(move.step, ts.step);
    limit(ts.pos, ts.step, STAGEMAX);
  end;
'T', 't': { Toggle stage. }
  begin
    write('Enter starting position (default = ');
    writevalue(move.start, POWTS, VALDECTS, VALMAXTS, 'm');
    writeln('):');
    readvalue(move.start, POWTS);
    roundoff(move.start, ts.step);
    limit(move.start, STAGEMIN, STAGEMAX);
    write('Enter ending position (default = ');
    writevalue(move.stop, POWTS, VALDECTS, VALMAXTS, 'm');
    writeln('):');
    readvalue(move.stop, POWTS);
    roundoff(move.stop, ts.step);
    limit(move.stop, STAGEMIN, STAGEMAX);
    write('Enter time to wait (default = ');
    writevalue(move.wait, POWTS, VALDECTS, VALMAXTS, 's');
    writeln('):');
    readvalue(move.wait, POWTS);
    roundoff(move.wait, 1e-3);
    if move.wait < 0 then
      move.wait := 0;
    sum := move.start + move.stop; { Math trick: add limits to-
      gether; then can toggle between the values by subtracting
      ts.pos from it each time (see below). }
    writeln('Moving to starting position. Press RETURN when ready.');
    StageMove(move.start);
    readln;
    writeln('Press any key to exit.');
    repeat
      StageMove(sum - ts.pos);
      write(BELL);
      delay(round(move.wait * 1000));
    until keypressed;
    { Remove character from buffer: }
    if readkey = EXTENDED then
      readkey;
  end;
'X', 'x': { Send disengage command (#EA) }
  begin
    writeln('#EA');
    write(COM2, ACK);
      delay(ACKDELAY);
      write(COM2, '#EA' + LF);
    end;
  ESC: { Exit. }
    begin
      write(COM2, ACK);
      exitflag := TRUE;
    end;
  end;
  until exitflag = TRUE;
  DrawAll;
end;

function StageDelay(r : real) : longint;
{ Calculates delay time for stage to move to position r from ts.pos.
  If ts.pos = 0 (always case at startup, and rarely thereafter), re-
  turns -delay which is a flag to have program wait for user input to
  know when stage has stopped moving. Value returned is in 100th of a
  second.}
var
  temp : longint;
begin
  temp := round(abs(r - ts.pos) * ts.wait.sl) + ts.wait.int; { Empirical
    formula. }
  if ts.pos > 0 then
    stagedelay := temp
  else
    stagedelay := -temp;
end;

function StageLookup(r : real) : real;
{ Looks up r in lookup table ts.wob.lookup to find corresponding
  position to submit to stage. }
var
  i, j, k : integer;
  ri, rj : real;
  rdiv : real;
begin
  if ts.wob.per < TS_STEP_MIN then
  { Do nothing if period is 0 (prevents /0 error). }
  begin
    stagelookup := r;
    exit;
  end;
  { Calculate integer divisor and modulus of lookup position. }
  rdiv := ts.wob.per * (trunc(r / ts.wob.per - ts.wob.ph + 1) - 1);
    { + 1 ensures arg of trunc function is positive. }
  r := ts.wob.per * frac(r / ts.wob.per - ts.wob.ph + 1);
   { + 1 ensures arg of frac function is positive. No need to add back
     since frac only keeps fraction anyway! }
  { Initialize positions. }
  i := 0;
  j := TS_LOOKUP_MAX;
  { Use binary search to find closest position... }
  while j - i > 1 do
  begin
    k := (i + j) div 2;
    if r < ts.lookup[k] then
      j := k
    else
      i := k;
  end;
  { ...and use interpolation to calculate fractional index position. }
```

```
  ri := ts.lookup[i];
  rj := ts.lookup[j];
  stagelookup := rdiv + ((i * (rj - r) + j * (r - ri)) / (rj - ri) /
    TS_LOOKUP_MAX + ts.wob.ph) * ts.wob.per;
end;

procedure StageMove(r : real);
{ Move stage to position r (m) and update ts.pos. }
const
  LF = #10; { For some reason compiler needs this defined again. }
var
  command : bufstring; { General string. }
begin
  if (r < STAGEMIN) or (r > STAGEMAX) then
    exit;
  write(COM2, ACK);
{  delay(ACKDELAY); }
  { StageLookup: Correct position using lookup table. ts.step: alter
    number of steps sent to ts based on current stepsize. }
  command := makestringint(round(StageLookup(r) / ts.step));
  write(COM2, '$CAD(' + command + ')' + LF);
  ts.pos := r;
end;

procedure StageMoveDelay(w : integer; r : real);
{ Move stage to position r. If ts.pos = 0 (the case when program is
  first started), automatically waits for user input before continuing.
  Changes wave[w]^.scan.starttime and steptime so program can stop after
  proper time. }
begin
  with wv.ls[w]^ do
  begin
    Time(scan.starttime);
    scan.steptime := StageDelay(r);
    StageMove(r);
    if scan.steptime < 0 then { Flag that user must tell when stage
      done moving. }
    begin
      com_wr(com_wr_MOVINGSTAGE, COLORHL);
      if readkey = EXTENDED then
        readkey;
      scan.steptime := 0; { No need to wait. }
    end;
  end;
end;

procedure StageMoveWait(r : real);
{ Moves stage and waits for calculated length of time. }
var
  del : longint;
  stoptime, t : longint;
begin
  del := 10 * StageDelay(r); { Delay in ms. }
  StageMove(r);
  if del < 0 then { Flag to get user's help. }
  begin
    com_wr(com_wr_MOVINGSTAGE, COLORHL);
    if readkey = EXTENDED then
      readkey;
  end
  else
  { Do a loop rather than using delay function, to allow user to stop
    early if we've got an unnaturally long wait. }
  begin
    time(stoptime);
    inc(stoptime, del div 10);
    repeat
      time(t);
    until keypressed or (t > stoptime);
    if keypressed then { Empty buffer. }
      if readkey = EXTENDED then
        readkey;
  end;
{    delay(del); }
end;

procedure SystemControl;
{ Allow control of some system features (mostly for debugging purposes). }
var
  exitflag : boolean;
  r : real;
  w : integer;
  l1, l2 : longint;
begin
  TextMode;
  exitflag := false;
  repeat
    writeln;
    writeln('System menu.');
    writeln('(A) Turn device reading on/off.');
    writeln('(B) Toggle debug flag.');
    writeln('(C) Print available memory.');
    writeln('(D) Print AD.ls[].on array.');
    writeln('(E) Print xh info on current screen.');
    writeln('(F) Print some info on current wave.');
    writeln('(G) Write graphics info for current screen.');

    writeln('(X) Exit.');
    case readkey of
    'a', 'A':
      begin
        dev_rd := 1 - dev_rd;
        writeln('dev_rd = ', dev_rd);
        case dev_rd of
        0:
          begin
            write(COM2, ACK);
            close(COM2);
          end;
        1:
          begin
            assign(COM2, 'COM2');
            rewrite(COM2);
            Init_ts;
          end;
        end;
      end;
    'b', 'B':
      begin
        debug := 1 - debug;
        writeln('Debug = ', debug);
      end;
    'c', 'C':
      begin
        writeln('Memavail = ', memavail, ' maxavail = ', maxavail);
        writeln('sizeof(sc) = ', sizeof(sc));
```

```
    writeln('sizeof(sc.ls[]) = ', sizeof(sc.ls[1]));
    writeln('sizeof(wv) = ', sizeof(wv));
    writeln('sizeof(wv.ls[]^) = ', sizeof(wv.ls[1]^));
    writeln('sizeof(com) = ', sizeof(com));
  end;
'd', 'D':
  for w := 1 to AD_MAX do
    writeln(w, ' ', ad.ls[w].on);
'e', 'E':
  with sc.ls[sc.cur].gr do
  begin
    writeln('xh.u[1] ', xh.u[1]);
    writeln('xh.u[2] ', xh.u[2]);
    writeln('xh.v[1] ', xh.v[1]);
    writeln('xh.v[2] ', xh.v[2]);
    writeln('xh.x[1] ', xh.x[1]);
    writeln('xh.x[2] ', xh.x[2]);
    writeln('xh.y[1] ', xh.y[1]);
    writeln('xh.y[2] ', xh.y[2]);
  end;
'f', 'F':
  if wv.num > 0 then
    with wv.ls[wv.cur]^ do
    begin
      writeln('vmin ', vmin);
      writeln('vmax ', vmax);
    end;
'g', 'G':
  with sc.ls[sc.cur].gr do
  begin
    writeln('cursorp ', cursorp);
    writeln('cursoru ', cursoru);
    writeln('cursorvisible ', cursorvisible);
    writeln('cursorx ', cursorx);
    writeln('cursory1 ', cursory1);
    writeln('cursory2 ', cursory2);
    writeln('maxxnums ', maxxnums);
    writeln('maxynums ', maxynums);
    writeln('ullim ', ullim);
    writeln('ulnorm ', ulnorm);
    writeln('ulnum ', ulnum);
    writeln('u2lim ', u2lim);
    writeln('u2norm ', u2norm);
    writeln('u2num ', u2num);
    writeln('ustep ', ustep);
    writeln('vllim ', vllim);
    writeln('vlnum ', vlnum);
    writeln('v2lim ', v2lim);
    writeln('v2num ', v2num);
    writeln('vstep ', vstep);
    waitkey;
    writeln('xaxisdt ', xaxisdt);
    writeln('xaxismode ', xaxismode);
    writeln('xdecimals ', xdecimals);
    writeln('xfullmode ', xfullmode);
    writeln('xlabelstring ', xlabelstring);
    writeln('xon ', xon);
    writeln('xpower ', xpower);
    writeln('xunits ', xunits);
    writeln('yaxismode ', yaxismode);
    writeln('ydecimals ', ydecimals);
    writeln('yfullmode ', yfullmode);
    writeln('ylabelstring ', ylabelstring);
    writeln('yon ', yon);
    writeln('ypower ', ypower);
    writeln('yunits ', yunits);
    waitkey;
  end;
'x', 'X': exitflag := true;
EXTENDED: readkey;
end;
until exitflag;
UpdateAll;
end;

procedure tekset_rd(ch : integer);
{ Reads settings from tek scope and saves in array teksetting[]. Reads
  status of all settings which are changed during data acquisition. ch
  is channel # to use. }
var
  s : bufstring;

  procedure set_rec(s : bufstring);
  { Grunt work of recording settings in array. }
  var
    i : integer;
  begin
    tekwrite_ver(s);
    mread(tek, sizeof(s) - 1);
    tekset.s[tekset.num] := '';
    for i := 1 to ibcnt - 1 do { disregard LF char. }
      tekset.s[tekset.num] := tekset.s[tekset.num] + ibbuf[i];
    inc(tekset.num);
    { Check to ensure do not overrun maximum: }
    if tekset.num > TEKSET_MAX then
    begin
      com_wr('Exceeded TEKSET_MAX.', COLORHL);
      dec(tekset.num);
    end;
  end;

begin
  mwrite(tek, 'head on', true); { Turn on verbose mode so we get entire
    commands sent upon querying. }
  tekset.num := 1;
  set_rec('acq:state?');
  tekwrite_ver('acq:state 0'); { Turn off so commands execute faster. }
  set_rec('select:control?'); { Read current channel being
    controlled. }
  set_rec('hor:delay:time:runsafter?');
  set_rec('hor:delay:scale?');
  set_rec('hor:main:scale?');
  set_rec('hor:mode?');
  set_rec('hor:trig:pos?');
  set_rec('hor:recordlength?');
  s := 'ch' + chr(ord('0') + ch);
  set_rec(s + ':offset?');
  set_rec(s + ':pos?');
  set_rec('acq:stopafter?');
  set_rec('acq:numavg?');
  set_rec('acq:mode?');
  set_rec('hor:fittoscreen?');
  set_rec('zoom:state?');
  tekwrite_ver('head off'); { Promptly shut off verbose mode. }
end;
```

```
procedure tekset_wr;
{ Writes recorded settings back to tek scope. }
var
  i : integer;
begin
  { Stop acquisition if in progress. }
  mwrite(tek, 'acq:state 0', true);
  { Write warning to Tek screen: }
{  tekwrite('message:box 74,209,474,259');
  tekwrite('message:show "Restoring settings. Please wait."');
  tekwrite('message:state on'); }
  { Restore settings in reverse order: }
  for i := tekset.num downto 1 do
    tekwrite(tekset.s[i]);
  { Erase message box: }
{  tekwrite('message:state off'); }
end;

function TEK_TIMEPERPT_to_code(timeperpt : real) : integer;
{ Find closest code to timeperpt. }
var
  i : integer;
begin
  for i := TEK_TIMEPERPT_MIN to TEK_TIMEPERPT_MAX - 1 do
    if (TEK_TIMEPERPT[i] <= timeperpt) and (timeperpt < TEK_TIMEPERPT[i
      + 1] * 0.999) then
    begin
      TEK_timeperpt_to_code := i;
      exit;
    end;
  TEK_timeperpt_to_code := TEK_TIMEPERPT_MAX;
end;

procedure tekwrite(wbuf : bufstring);
{ Waits to see if tek is free, then calls mwrite. }
begin
  repeat
    mwrite(tek, 'busy?', true);
    mread(tek, 2);
  until ibbuf[1] = '0';
  mwrite(tek, wbuf, true);
end;

procedure tekwrite_ver(wbuf : bufstring);
{ Waits to see if tek is free, then calls mwrite. }
begin
  repeat
    mwrite(tek, 'busy?', true);
    mread(tek, 8);
  until ibbuf[7] = '0';
  mwrite(tek, wbuf, true);
end;

procedure TextMode;
{ Exits graphics mode }
begin
  restorecrtmode;
  TextColor(WHITE);
  sc.mode := sc_mode_TX;
end;

procedure TidyUp;
var
  i : integer;
begin
  { Free memory for virtual oscilloscope bitmap buffer. }
  FreeMem(osc.bit, imagesize(sc.bdy[1].x, 0, sc.bdy[2].x, 0));
  { Free memory for color list. }
  FreeMem(com.tx.col, 2 * com.tx.num.y);
  { Free memory for buffer. }
  FreeMem(com.tx.buf, com.tx.num.x * com.tx.num.y);
  for i := MAXWAVES downto 1 do
    Dispose(wv.ls[i]); { Free wave memory. }
  for i := sc_MAX downto 1 do
  { Deallocate previously reserved memory, in reverse order. }
    with sc.ls[i].gr do
    begin
      FreeMem(xh.bitmap.y[3], imagesize(sc.bdy[1].x, 0, sc.
        bdy[2].x, 0));
      FreeMem(xh.bitmap.y[2], imagesize(sc.bdy[1].x, 0, sc.
        bdy[2].x, 0));
      FreeMem(xh.bitmap.y[1], imagesize(sc.bdy[1].x, 0, sc.
        bdy[2].x, 0));
      FreeMem(xh.bitmap.x[3], imagesize(0, sc.bdy[1].y, 0, sc.
        bdy[2].y));
      FreeMem(xh.bitmap.x[2], imagesize(0, sc.bdy[1].y, 0, sc.
        bdy[2].y));
      FreeMem(xh.bitmap.x[1], imagesize(0, sc.bdy[1].y, 0, sc.
        bdy[2].y));
    end;
  FreeMem(bitmap, imagesize(0, 0, 0, CURSORLENGTH)); { Dispose of cursor
    bitmap memory. }
  if dev_rd = 1 then
  begin
    write(COM2, ACK); { Send final ACK code. }
    close(com2); { Close translation stage channel. }
  end;
  CloseGraph; { free graphics memory (graph unit procedure) and return to text
    mode. }
end;

procedure Time(var nowtime : longint);
var
  hour, minute, second, sec100 : word;
begin
  GetTime(Hour, Minute, Second, Sec100);
  nowtime := hour * 60 * 60 * 100 + minute * 60 * 100 + second * 100 +
    sec100;
end;

function TimebaseToCode(r : real) : integer;
{ Finds code corresponding to smallest timebase. }
var
  i : integer;
begin
  r := r * (1 + SMALL); { Make little bigger to ensure hit minimum. }
  for i := MCS_TIMEPERPT_MIN to MCS_TIMEPERPT_MAX - 1 do
    if (MCS_TIMEPERPT[i] <= r) and (r < MCS_TIMEPERPT[i + 1]) then
    begin
      TimebaseToCode := i;
      exit;
    end;
  TimebaseToCode := MCS_TIMEPERPT_MAX;
end;

procedure ToggleAddWavesMode(w : integer);
```

```
{ If addwaves mode is off, turns on for wave w; if already on, then
  turn off. }
var
  i : integer;
begin
  if wv.num = 0 then
    exit;
  if addwaves.mode = 1 then
  begin
    UpdateAddWaves; { Add for last time before quitting. }
    addwaves.mode := 0;
    UpdateVitals;
    Update(wv.ls[addwaves.w]^.screen); { Final screen update. }
  end
  else
  begin
    if wv.ls[w]^.scan.mode > 0 then
    begin
      com_wr('Cannot overwrite scan in progress!', COLORHL);
      exit;
    end;
    if wv.ls[w]^.datasaved = 0 then
      if com_wr_yn('Wave not saved. Overwrite') = 0 then
        exit;
    if (sc.ls[wv.ls[w]^.screen].gr.xaxismode <> XAXISMODE_POINTS) then
      com_wr('Not in points mode: no promises!', COLORMESS);
    addwaves.mode := 1;
    addwaves.w := w;
    { Find all screens with valid waves to add. }
    sc_sel_off;
    for i := 1 to wv.num do
      with wv.ls[i]^ do
        if (on = 1) and (par.dt = wv.ls[addwaves.w]^.par.dt)
          then
          sc.ls[screen].sel := 1;
    { Ensure all selected screens are in relative y axis mode. }
    for i := 1 to sc.num do
      with sc.ls[i] do
        if (sel = 1) and (gr.yaxismode = YAXISMODE_ABSOLUTE) then
          ToggleYAxisMode(i);
    { Update add waves screen since erased previous data. }
    UpdateVitals;
    Update(wv.ls[addwaves.w]^.screen);
  end;
end;

procedure ToggleCrosshairsMode;
{ Turn xh mode on or off. }
begin
  EraseCursor(sc.cur);
  sc.ls[sc.cur].gr.xh.mode := 1 - sc.ls[sc.cur].gr.xh.
    mode;
  UpdateCursor(sc.cur);
  DrawCursor(sc.cur);
  DrawCursorInfo;
end;

procedure ToggleCrosshairsWhich;
{ Switch which xh are manipulated by arrow keys. }
begin
  with sc.ls[sc.cur].gr do
    if xh.mode = 1 then
    begin
      xh.which := 3 - xh.which; { Switch between 1 and
        2. }
      EraseCursor(sc.cur);
      DrawCursor(sc.cur);
    end;
end;

procedure ToggleSaveMode;
{ Toggle between saving data multiplied by yscale or not. }
begin
  globalSaveMode := 1 - globalSaveMode;
  case globalSaveMode of
    0 : com_wr('Regular save mode.', COLORMESS);
    1 : com_wr('Normalized save mode.', COLORMESS);
  end;
end;

procedure ToggleXAxisMode(scr : integer; mode : integer);
{ Toggle x axis among three display modes: XAXISMODE_POINTS,
  XAXISMODE_NORMAL and XAXISMODE_CONVERT. When switching from one to the
  other, u-space limits will need to be changed. }
var
  p : integer;                          { Point index. }
  p1 : integer;                          { Point index. }
  p2 : integer;                          { Point index. }
  w : integer; { Wave pointer. }
begin
  { Find visible wave on screen scr. }
  w := Findwave(scr);
  { Now take a look at situation. }
  with sc.ls[scr].gr do
  begin
    if w = 0 then
      xfullmode := 1
    else
    begin
      { Find points corresponding to screen limits. }
      if xaxismode = XAXISMODE_POINTS then
      begin
        p1 := round(u1lim);
        p2 := round(u2lim);
        if cursorvisible = 1 then
          cursorp := round(cursoru);
      end
      else
      begin
        p1 := UtoP(w, u1lim, 1);
        p2 := UtoP(w, u2lim, -1);
        if cursorvisible = 1 then
          cursorp := UtoP(w, cursoru, 0);
      end;
      { Special handling for movement into energy units: }
      if xaxisdt = dt_ELE then
      begin
        { Switch limits. }
        if xaxismode = XAXISMODE_CONVERT then
        begin
          p := p1;
          p1 := p2;
          p2 := p;
        end;
        if mode = XAXISMODE_CONVERT then
        begin
```

```
        { Switch order of limits. }
        p := p1;
        p1 := p2;
        p2 := p;
      end;
    end;
  end;
  { Change mode. }
  xaxismode := mode;
  { Change dt to current wave if on same screen. }
  if w > 0 then
  begin
    if wv.ls[w]^.screen = sc.cur then
      xaxisdt := wv.ls[w]^.par.dt;
    { Assign minimum useful point. }
    if p1 < wv.ls[w]^.pmin then
      p1 := wv.ls[w]^.pmin;
    if cursorp < wv.ls[w]^.pmin then
      cursorp := wv.ls[w]^.pmin;
    cursoru := PtoU(w, cursorp, xaxismode); { New cursor position. }
  end;
  { Prevent program from crashing (this is sort of a cop-out!) }
  if (xfullmode = 0) and ((p1 = 0) or (p2 = 0)) then
    xfullmode := 1;
  { Calculate new unit-space limits. }
  if xfullmode = 0 then
  begin
    ullim := PtoU(w, p1, xaxismode);
    u2lim := PtoU(w, p2, xaxismode);
  end;
  { Calculate other vars and redraw. }
  UpdateVitals;
  Update(scr);
end;

procedure ToggleYAxisMode(scr : integer);
begin
  sc.ls[scr].gr.yaxismode := 1 - sc.ls[scr].gr.yaxismode;
  UpdateVitals;
  Update(scr);
end;

procedure ToggleYOffsetRescale;
{ Toggle flag to rescale wv.ls[]^.par.yoffset when wv.ls[]^.par.yscale is
  changed. }
begin
  yoffsetrescale := 1 - yoffsetrescale;
  if yoffsetrescale = 0 then
    com_wr('Y offset rescale mode OFF.', COLORMESS)
  else
    com_wr('Y offset rescale mode ON.', COLORMESS);
end;

procedure tx_dr(tx : tx_type_p);
{ Draw text screen. Algorithm is to create a string for each line, then
  call graphics routine. For speed, we copy characters directly into s,
  which requires some pointer tricks. }
var
  p : ^chararray; { Pointer to chars in buffer. }
  s : bufstring; { Holder for each line. }
  x, y : integer; { Char counters. }
begin
  if sc.mode <> sc_mode_GR then
    exit;
{  SetTextJustify(LEFTTEXT, TOPTEXT);
  SetTextStyle(DEFAULTFONT, HORIZDIR, 1); }
  { Clear screen. }
  ClearArea(tx^.bdy[1].x, tx^.bdy[1].y, tx^.bdy[2].x, tx^.bdy[2].y);
  { Fool Pascal into thinking s has length = tx.num.x. }
  p := @s;
  p^[1] := chr(tx^.num.x);
  for y := 1 to tx^.num.y do
  begin
    { Point to start of current line. }
    p := @(tx^.buf^[(y - 1) * tx^.num.x + 1]);
    { Copy line into s. }
    for x := 1 to tx^.num.x do
      s[x] := p^[x];
    { Print line to screen. }
    SetColor(tx^.col^[y]);
    OutTextXY(tx^.bdy[1].x, tx^.bdy[1].y + (y - 1) * textsize, s);
  end;
end;

procedure tx_scr_up(tx : tx_type_p);
{ Scroll text screen up one line. }
var
  i : integer;
begin
  { Scroll up character buffer. }
  for i := 1 to tx^.num.x * (tx^.num.y - 1) do
    tx^.buf^[i] := tx^.buf^[i + tx^.num.x];
  { Blank last line. }
  FillChar(tx^.buf^[i + 1], tx^.num.x, ' ');
  { Scroll up color list. }
  for i := 1 to tx^.num.y - 1 do
    tx^.col^[i] := tx^.col^[i + 1];
  { Make last line user input color. }
  tx^.col^[i + 1] := COLORUSER;
  { Redraw screen. }
  tx_dr(tx);
end;

procedure tx_wr(tx : tx_type_p; s : string; col : word; scr : integer);
{ Print string s to tx screen in color col. Cursor moves to left side of
  following line unless scr = 0, in which case cursor is left at end of
  string. (Screen still scrolls as needed to fit entire string on it, but
  messages larger than size of screen are truncated). If col = COLORHL
  (highlight color), a beep will also sound. }
var
  i : integer; { Counter. }
  nch : integer; { Max. number of chars allowed. }
  nstart : integer; { Char to start with on screen. }
  nscr : integer; { Number of lines to scroll. }
  p : ^chararray;
begin
  { Calculate starting character; always starts on lefthand side. }
  nstart := tx^.num.x * (tx^.cur.y - 1) + 1;
  { Calculate max. number of chars allowed in message. Note this is 1
    line short of box size, unless scr = 0 -- must allow to scroll up
    1 line. Extra subtraction for scr = 0 is to force scroll if cursor
    will end up on next line. Truncate message if needed. }
  nch := tx^.num.x * (tx^.num.y - scr) - 1 + scr;
  if length(s) > nch then
    s := copy(s, 1, nch - 2) + '..'; { Little flag to user to indicate
```

```
      line was truncated. }
  { Scroll screen to fit message, if needed. Second -1 is to not scroll
    screen extra line if we JUST fill last line. }
  nscr := better_div(nstart - 1 + length(s) - nch - 1, tx^.num.x) + 1;
  if nscr > 0 then
  begin
    for i := 1 to nscr do
      tx_scr_up(tx);
    dec(tx^.cur.y, nscr);
    dec(nstart, nscr * tx^.num.x);
  end;
  { Set pointer to beginning of current line. }
  p := @(tx^.buf^[nstart]);
  { Copy message to buffer. }
  for i := 1 to length(s) do
    p^[i] := s[i];
  { Change color on affected lines. }
  for i := 0 to better_div(length(s) - 1, tx^.num.x) do
    tx^.col^[tx^.cur.y + i] := col;
  { Text screen handling: Only redraw graphics if message is "important"
    -- that is, using COLORHL to draw. Otherwise, don't redraw (but
    message still appears in buffer). }
  if (sc.mode <> sc_mode_GR) and (col = COLORHL) then
  begin
    sc.mode := sc_mode_TX; { Allow screen to be redrawn. }
    DrawAll;
  end;
  tx_dr(tx);
  { Highlight feature. }
  if col = COLORHL then
    write(BELL);
  { Update cursor location. }
  case scr of
  0 :
    begin
      tx^.cur.x := length(s) mod tx^.num.x + 1;
      inc(tx^.cur.y, length(s) div tx^.num.x);
    end;
  1 :
    begin
      tx^.cur.x := 1;
      inc(tx^.cur.y, better_div(length(s) - 1, tx^.num.x) + 1);
    end;
  end;
  { Make current line default user color. }
  tx^.col^[tx^.cur.y] := COLORUSER;
end;

procedure tx_wr_ch(tx : tx_type_p; c : char);
{ Replaces character c at current cursor location in text buffer, UpdateAll
  screen. }
begin
  { Erase old character by writing in background color: }
  SetColor(GetBkColor);
  OutTextXY(tx^.bdy[1].x + (tx^.cur.x - 1) * textsize, tx^.bdy[1].y +
    (tx^.cur.y - 1) * textsize, tx^.buf^[tx^.cur.x + (tx^.cur.y - 1) *
    tx^.num.x]);
  { Replace character and print to screen in current line color. }
  tx^.buf^[tx^.cur.x + (tx^.cur.y - 1) * tx^.num.x] := c;
  SetColor(tx^.col^[tx^.cur.y]);
  OutTextXY(tx^.bdy[1].x + (tx^.cur.x - 1) * textsize, tx^.bdy[1].y +
    (tx^.cur.y - 1) * textsize, c);
end;
```

```
end.
```

4.6. fpesuz.pas

```
unit FpesUZ;
{ History of modifications (please add to BOTTOM of list!):

  Version 1: Begun 2jun94 BJG.

  Procedures and functions beginning with Y through Z for the program FPES.
  See FPES (at appropriate version) for more specific program modification
  notes.
}

interface

uses
  FpesVar;

function UnitPrefix(logpower : integer) : bufstring;
procedure Update(scr : integer);
procedure UpdateAll;
procedure UpdateAddWaves;
procedure UpdateBg;
procedure UpdateCORLimits(par_ptr : par_type_ptr);
procedure UpdateCursor(scr : integer);
procedure UpdateDtCompatibility;
procedure UpdateFilenames(newdir : bufstring);
procedure UpdateMC;
procedure UpdatePMin;
procedure UpdateSel;
procedure UpdateVitals;
procedure UpdateVLimits;
procedure UpdateXAxis(scr : integer);
procedure UpdateYAxis(scr : integer);
function UtoP(w : integer; u : real; direction : integer) : integer;
function UtoX(u : real; scr : integer) : real;
procedure val_i(s : bufstring; var i : integer);
procedure val_r(s : bufstring; var r : real);
function VtoY(v : real; scr : integer) : real;
procedure waitkey;
function WaveInfo(w : integer; l : integer) : bufstring;
function WavesExist : integer;
function weight(x , a : real) : real;
procedure writeengnotation(r : real);
procedure writeln_ctr(s : string; pos : integer);
procedure writevalue(r, p : real; dec, max : integer; u : string);
procedure writevalueunitless(r, p : real; dec, max : integer);
procedure ws_df;
procedure ws_init;
procedure ws_rd(s : bufstring);
procedure ws_sv(s : bufstring);
procedure wv_sel(code : integer);
procedure wv_sel_off;
procedure XFull(scr : integer);
function XToTheY(x, y : real) : real;
function XtoU(x : longint; scr : integer) : real;
procedure YFull(scr : integer);
function YtoV(y : longint; scr : integer) : real;
```

```
procedure YScaleChange(r : real);
procedure YScaleChangeSign;
procedure Zoom;

implementation

uses
  crt, dos, FpesCom, FpesAI, FpesJR, FpesST, graph, Keys, TPDecl;

function UnitPrefix(logpower : integer) : bufstring;
{ Returns unit prefix symbol corresponding to exponent logpower. If logpower
  is outside range of MINPOWER .. MAXPOWER, or it's not a factor of 3,
  returns string '10^' + logpower. }
begin
  case logpower of
  -18: UnitPrefix := 'a';
  -15: UnitPrefix := 'f';
  -12: UnitPrefix := 'p';
   -9: UnitPrefix := 'n';
   -6: UnitPrefix := 'u';
   -3: UnitPrefix := 'm';
    0: UnitPrefix := '';
    3: UnitPrefix := 'k';
    6: UnitPrefix := 'M';
    9: UnitPrefix := 'G';
   12: UnitPrefix := 'T';
  else
    UnitPrefix := PowerOfTenPrefix(logpower) + ' '; { Extra space for clari-
      ty (e.g. "10^-33 J"). }
  end;
end;

procedure Update(scr : integer);
{ Update only screen scr. Must call UpdateVitals beforehand! }
var
  s : bufstring;
  w : integer; { Wave counter. }
begin
  { Calculate new u and v limits if in full view mode for that axis: }
  XFull(scr);
  YFull(scr);
  { Calculate screen coordinates (x and y) from u and v information: }
  UpdateXAxis(scr);
  UpdateYAxis(scr);
  { Update location of cursor, given wv.cur and cursoru information: }
  UpdateCursor(scr);
  { Draw new screen: }
  DrawScreen(scr);
end;

procedure UpdateAll;
{ Calculates everything needed for proper screen display. Should be called
  after every command which changes some aspect of the screen display or wave
  data, though if you are clever you can call only a subset of update rou-
  tines. }
var
  scr : integer; { Screen counter. }
begin
  UpdateVitals;
  for scr := 1 to sc.num do
  begin
    { Calculate new u and v limits if in full view mode for that axis: }
    XFull(scr);
    YFull(scr);
    { Calculate screen coordinates (x and y) from u and v information: }
    UpdateXAxis(scr);
    UpdateYAxis(scr);
    { Update location of cursor, given wv.cur and cursoru information: }
    UpdateCursor(scr);
  end;
  { Draw new screen: }
  DrawAll;
end;

procedure UpdateAddWaves;
{ Adds together all visible waves of same dt as addwaves.w,
  weighted by their par.yscale and par.yoffset values, and places in
  wave addwaves.w. }
var
  j : integer;
  w : integer;
begin
  if addwaves.mode = 0 then
    exit;
  FillChar(wv.ls[addwaves.w]^.data, sizeof(wv.ls[addwaves.w]^.data),
    0);
  FillChar(wv.ls[addwaves.w]^.tmp, sizeof(wv.ls[addwaves.w]^.tmp), 0);
  wv.ls[addwaves.w]^.datasaved := 0;
  for w := 1 to wv.num do
    if (w <> addwaves.w) and (wv.ls[w]^.on = 1) and (wv.ls[w]^.par.
      dt = wv.ls[addwaves.w]^.par.dt) then
      for j := 1 to wv.ls[w]^.par.pt do
        wv.ls[addwaves.w]^.data[j] := wv.ls[addwaves.w]^.data[j] +
          wv.ls[w]^.par.yscale * wv.ls[w]^.data[j] + wv.ls[w]^.par.yoffset;
end;

procedure UpdateBg;
{ If _bs.bg > 0, go thru waves to ensure that there are bs waves in
  memory, and that _bs.bg points to one. Otherwise, sets _bs.bg to
  0. }
var
  i : integer;
begin
  if (_bs.bg > wv.num) or (_bs.bg < 0) then
    _bs.bg := 0;
  if _bs.bg = 0 then
    exit;
  { Check that bg points to a bs ELE wave; otherwise, reset and turn
    off bs mode for all waves. }
  with wv.ls[_bs.bg]^ do
    if (par.dt <> DT_ELE) or (par.ele.bs.mode = 0) then
    begin
      _bs.bg := 0;
      for i := 1 to wv.num do
        with wv.ls[i]^ do
          if par.dt = dt_ELE then
            par.ele.bs.mode := 0;
      DrawWaveData;
      exit;
    end;
  { Go thru waves until we find a bs ELE wave. }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if (par.dt = dt_ELE) and (par.ele.bs.mode > 0) then
        exit;
  { No bs waves loaded; turn off bg flag. }
```

```
  _bs.bg := 0;
end;

procedure UpdateCORLimits(par_ptr : par_type_ptr);
{ Update min & max limits of COR wave based on t0 and time per point. }
begin
  with par_ptr^ do
  with cor.ts do
  begin
    start := t0 - step * (pt_gl - 1) / 2;
    stop := t0 + step * (pt_gl - 1) / 2;
    limit(start, STAGEMIN, STAGEMAX);
    limit(stop, STAGEMIN, STAGEMAX);
    { Recalculate pt_gl in case limits were changed. }
    pt_gl := round((stop - start) / step) + 1;
  end;
end;

procedure UpdateCursor(scr : integer);
{ Updates cursor on screen scr. Two functions:
  1. If xh.mode = 0: Provided cursor is on a visible wave, uses
  wv.cur and cursoru to calculate corresponding cursorp, cursorx, cursory1,
  cursory2, making sure cursor is on screen and on a legitimate point. If
  cursor cannot be displayed (wave is completely off-screen), sets cursor-
  visible to 0.
  2. If xh.mode = 1: Calculate u's and v's corresponding to x and y
  positions of xh. Note x's and y's never change when changing axis
  scales or modes, or switching in and out of xh mode. }
var
  i : integer; { Generic counter. }
  u : real; { Temporary u value. }
begin
{  if wv.num = 0 then
    exit; }
  with sc.ls[scr].gr do
    if xh.mode = 0 then
    begin
      if wv.num = 0 then
      begin
        cursorvisible := 0;
        exit;
      end;
      with wv.ls[wv.cur]^ do
      begin
        { Make sure we're on a visible wave on same screen: }
        if (on = 0) or (screen <> scr) then
        begin
          cursorvisible := 0;
          exit;
        end;
        { Make sure cursoru is on visible part of x axis: }
        if cursoru < u1lim then
          cursoru := u1lim
        else if cursoru > u2lim then
          cursoru := u2lim;
        { Seek closest point of wave which corresponds in u value to cursoru, and
          see if this is visible on the screen; otherwise, turn cursor off: }
        cursorp := UtoP(wv.cur, cursoru, 0);
        u := PtoU(wv.cur, cursorp, xaxismode);
        if (u < u1lim) or (u > u2lim) then
        begin
          cursorvisible := 0;
          exit;
        end;
        { Calculate remaining cursor variables: }
        cursoru := u;
        cursorx := round(UtoX(cursoru, scr));
        cursory2 := round(PtoY(wv.cur, cursorp));
        { Constrain cursor y position to stay on screen: }
        if cursory2 < yaxis.y1 then
          cursory2 := yaxis.y1 + CURSORLENGTH
        else if cursory2 > yaxis.y2 - 1 then
          cursory2 := yaxis.y2 - 1;
        cursory1 := cursory2 - CURSORLENGTH; { Make cursor extend upward from
          point. }
        if cursory1 < yaxis.y1 then
        begin
          cursory1 := cursory2;
          cursory2 := cursory1 + CURSORLENGTH; { If tip is above top of plot area,
            just make it point downward. }
        end;
        cursorvisible := 1;
        UpdateMC;
      end;
    end
    else with xh do
    begin
      { Calculate position of center xh: }
      x[3] := (x[1] + x[2]) div 2;
      y[3] := (y[1] + y[2]) div 2;
      { Calculate u- and v-space values of xh: }
      for i := 1 to 3 do
      begin
        u[i] := XtoU(x[i], scr);
        v[i] := YtoV(y[i], scr);
      end;
    end;
end;

procedure UpdateDtCompatibility;
{ This mouthful of a procedure simply turns off waves whose datatypes
  don't match their screen's datatype (if in NOR or CONV mode). }
var
  i : integer;
begin
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      with sc.ls[screen].gr do
        if (xaxismode <> XAXISMODE_POINTS) and (par.dt <> xaxisdt) then
          on := 0;
end;

procedure UpdateFilenames(newdir : bufstring);
{ Changes recorded filename when dir changes. Should not be called as
  part of standard Update -- slows system! }
var
  i, w : integer;
begin
  for w := 1 to wv.num do
    with wv.ls[w]^ do
    begin
      if not FullPath(par.fn) then
        par.fn := dir + par.fn;
      i := FilenameStart(par.fn);
      if copy(par.fn, 1, i - 1) = newdir then
        par.fn := copy(par.fn, i, length(par.fn));
```

```
    end;
  dir := newdir;
end;

procedure UpdateMC;
{ According to cursor position, calculate all eligible mass combina-
  tions according to mc parameters. }
var
  abort : boolean; { Flag to kill current combination if multiplicity
    limits are exceeded. }
  i : integer;
  mass : real;
  r : real;

  procedure mc_wr;
  { Write element combination to screen and its mass. }
  var
    i : integer;
    mass : real;
    s2, s3 : bufstring;
  begin
    if abort then
      exit;
    if mc.s <> '' then
      mc.s := mc.s + ', '; { Add new element. }
    s2 := '';
    mass := 0;
    for i := 1 to mc.num do
      with mc.ls[i] do
        if num > 0 then
        begin
          if num > 1 then
            str(num, s3)
          else
            s3 := '';
          s2 := s2 + s + s3;
          mass := mass + m * num;
        end;
    mc.s := mc.s + makestringint(round(mass)) + ' ' + s2;
  end;

  procedure make_comb(i : integer);
  { Makes combination of elements of starting with element i. Remaining
    mass is r. }
  var
    j : integer;
  begin
    abort := false;
    r := mass;
    { Subtract mass of assigned parts of molecule: }
    for j := 1 to i - 1 do
      r := r - mc.ls[j].m * mc.ls[j].num;
    { Find max. number of each element remaining: }
    for j := i to mc.num do
      if (abort = false) and (r > -mc.sens) then
        with mc.ls[j] do
        begin
          num := trunc((r + mc.sens) / m); { Find highest valid
            multiple. }
          if num > max then { Force to be <= max. }
            num := max;
          if num < min then
            abort := true
          else
            r := r - num * m; { Remove mass from r. }
        end;
    end;

  begin
    mc.s := '';
    if wv.num = 0 then
      exit;
    if wv.ls[wv.cur]^.par.dt <> dt_MAS then
      exit;
    { Find current cursor mass. }
    mass := PtoU(wv.cur, sc.ls[wv.ls[wv.cur]^.screen].gr.cursorp,
      xaxismode_CONVERT);
    { Begin iterative search for valid element combinations. }
    make_comb(1);
    if abs(r) <= mc.sens then
      mc_wr;
    i := mc.num - 1;
    { Now loop through other possibilities. }
    while i > 0 do
      with mc.ls[i] do
        if num > min then
        begin
          dec(num);
          make_comb(i + 1);
          if abs(r) <= mc.sens then
            mc_wr;
          i := mc.num - 1;
        end
        else
          dec(i);
  end;

  procedure UpdatePMin;
  { Calculate minimum meaningful point which should be displayed in each wave.
    Value is different depending on xaxismode and dt:

    XAXISMODE_POINTS, XAXISMODE_NORMAL: All points good; pmin := 1.
    XAXISMODE_CONVERT:
      dt_COR, dt_POW: All points good; pmin := 1.
      dt_ELE: pmin corresponds to first point with energy < par.
        user[PAR_E_LASEREV], the largest physically allowed energy.
      dt_IONS: pmin corresponds to first point with mass > 0.
  }
  var
    laserev : real;
    w : integer;
    xaxismode : integer;
  begin
    for w := 1 to wv.num do
    with wv.ls[w]^ do
    begin
      pmin := 1;
      if (sc.ls[screen].gr.xaxismode = XAXISMODE_CONVERT) and (par.dt
        = dt_ELE) then
      begin
        xaxismode := sc.ls[screen].gr.xaxismode;
        laserev := par.ele.cal.ev;
        while (pmin <= par.pt) and (PtoU(w, pmin, xaxismode) >
          laserev) do
          inc(pmin);
        if pmin > par.pt then { Shut off wave if no points are valid. }
```

```
        on := 0;
    end;
  end;
end;

procedure UpdateSel;
{ Update only those screens selected in sc.ls[].sel. }
var
  i : integer;
begin
  UpdateVitals;
  for i := 1 to sc.num do
    if sc.ls[i].sel = 1 then
      Update(i);
  sc_sel_off;
end;

procedure UpdateVitals;
{ Updates parameters for all waves which only have to be evaluated once
  per update. Designed to save time when calling Update for several
  screens. }
begin
  UpdateAddWaves;
  UpdateBg;
  UpdateDtCompatibility;
  UpdateFilenames(dir);
  UpdatePMin;
  UpdateVLimits;
  DrawVitals;
end;

procedure UpdateVLimits;
{ Find largest and smallest V values in each wave. }
var
  p : integer; { Point index. }
  v : real; { Temporary y value. }
  w : integer;
  xaxismode, yaxismode : integer;
begin
  for w := 1 to wv.num do
  with wv.ls[w]^ do
  begin
    xaxismode := sc.ls[screen].gr.xaxismode;
    yaxismode := sc.ls[screen].gr.yaxismode;
    vmin := PtoV(w, pmin, xaxismode, yaxismode);
    vmax := vmin;
    for p := pmin + 1 to par.pt do
    begin
      v := PtoV(w, p, xaxismode, yaxismode);
      if v < vmin then
        vmin := v;
      if v > vmax then
        vmax := v;
    end;
  end;
end;

procedure UpdateXAxis(scr : integer);
{ Calculates needed quantities when x limits change for screen scr. All
  that needs to be set before calling this routine is u1lim, u2lim,
  xaxismode, xaxisdt. }
var
  s : bufstring; { Generic string. }
begin
  with sc.ls[scr].gr do
  begin
    { Error prevention: }
    if u1lim = u2lim then
      if u1lim = 0 then
        u2lim := 1
      else
        u2lim := u1lim + SMALL * abs(u1lim);
  { Calculate axis numbers. }
    ustep := exp(LN_10 * rounddown(ln(u2lim - u1lim) / LN_10 - 2));
    { Begin guess with a power of 10, at least 2 orders of magnitude too small. }
    u1num := ustep * roundup(u1lim / ustep - SMALL); { Leftmost number is smal-
      lest which is larger than u1lim, and a multiple of ustep. }
    u2num := ustep * rounddown(u2lim / ustep + SMALL); { Rightmost number is
      largest which is smaller than u2lim, and a multiple of ustep. }
    while (u2num - u1num) / ustep + 1 > maxxnums do { if number of numbers
      exceeds maximum allowed. }
    begin
      ChangeStep(ustep, 1); { Increase step size. }
      u1num := ustep * roundup(u1lim / ustep - SMALL);
      u2num := ustep * rounddown(u2lim / ustep + SMALL);
    end;
  { Find largest multiple-of-3 power to units. }
    if abs(u1lim) > abs(u2lim) then
      xpower := Power(abs(u1lim))
    else
      xpower := Power(abs(u2lim));
  { Calculate number of decimals needed to display numbers (in current units). }
    xdecimals := -rounddown(ln(ustep) / LN_10 - Log10(xpower));
    if xdecimals < 0 then
      xdecimals := 0; { Correct for negative # of decimal places! }
  { Update x axis label and units. }
    case xaxismode of
    XAXISMODE_POINTS:
      begin
        xlabelstring := 'Points';
        xunits := PowerOfTenPrefix(round(Log10(xpower)));
        if xpower <> 1 then
          xlabelstring := xlabelstring + '/';
      end;
    XAXISMODE_NORMAL:
      begin
        xunits := UnitPrefix(round(log10(xpower))) + XUNITTYPE[xaxisdt,
          xaxismode];
        case xaxisdt of
        dt_COR: xlabelstring := 'Distance/';
        else xlabelstring := 'Time/';
        end;
      end;
    XAXISMODE_CONVERT:
      begin
        xunits := UnitPrefix(round(log10(xpower))) + XUNITTYPE[xaxisdt,
          xaxismode];
        case xaxisdt of
        dt_COR, dt_POW: xlabelstring := 'Time/';
        dt_ELE: xlabelstring := 'Energy/';
        dt_MAS: xlabelstring := 'Mass/';
        end;
      end;
    end;
  end;
end;
```

```
procedure UpdateYAxis(scr : integer);
{ Calculates needed quantities when y limits change for screen scr. All
  that needs to be set before calling this routine is v1lim, v2lim,
  yaxismode. }
var
  s : bufstring;
begin
  with sc.ls[scr].gr do
  begin
  { Error prevention: }
    if v1lim = v2lim then
      if v1lim = 0 then
        v2lim := 1
      else
        v2lim := v1lim + SMALL * abs(v1lim);
  { Calculate y axis numbers. }
    vstep := exp(LN_10 * rounddown(ln(v2lim - v1lim) / LN_10 - 2));
    { Begin guess with a power of 10, at least 2 orders of magnitude too small. }
    v1num := vstep * roundup(v1lim / vstep - SMALL);
    { Leftmost number is smallest which is larger than v1lim, and a multiple of
      vstep. }
    v2num := vstep * rounddown(v2lim / vstep + SMALL);
    { Rightmost number is largest which is smaller than v2lim, and a multiple of
      vstep. }
    while (v2num - v1num) / vstep + 1 > maxynums do { if number of numbers
      exceeds maximum allowed. }
    begin
      ChangeStep(vstep, 1); { Increase step size. }
      v1num := vstep * roundup(v1lim / vstep - SMALL);
      v2num := vstep * rounddown(v2lim / vstep + SMALL);
    end;

  { Find largest multiple-of-3 power to units. }
    if abs(v1lim) > abs(v2lim) then
      ypower := Power(abs(v1lim))
    else
      ypower := Power(abs(v2lim));

  { Calculate number of decimals needed to display numbers. }
    ydecimals := -rounddown(ln(vstep) / LN_10 - Log10(ypower));
    if ydecimals < 0 then
      ydecimals := 0; { Correct for negative # of decimal places! }

  { Update y axis label and units. }
    if yaxismode = YAXISMODE_ABSOLUTE then
      ylabelstring := 'Int.'
    else
      ylabelstring := 'Rel. Int.';
    if ypower <> 1 then
      ylabelstring := ylabelstring + '/';
    yunits := PowerOfTenPrefix(round(Log10(ypower)));
  end;
end;

function UtoP(w : integer; u : real; direction : integer) : integer;
{ Converts unit space x coordinate u to closest point in wave w, according to
  the criterion variable direction, which indicates which way to round:
   -1: Pick point with next lowest u value. If entire wave > u, return 0.
    0: Pick point with closest u value.
    1: Pick point with next highest u value. If entire wave < u, return 0.
  Usually, 1 is chosen when searching for the leftmost point which fits on
  the screen; -1 is chosen for the rightmost point. 0 is chosen when search-
  ing for closest point, as when moving between waves, or selecting a peak
  for normalization. }
var
  p : integer;                          { Point index. }
  p1 : integer;                         { Point with smallest allowed unit value.
                                          }
  p2 : integer;                         { Point with largest allowed unit value. }
  up : real;
begin
  with wv.ls[w]^ do
  begin
    { Establish limits to search. }
    if (sc.ls[wv.ls[w]^.screen].gr.xaxismode = XAXISMODE_CONVERT) and
      (par.dt = dt_ELE) then
    begin
      p1 := par.pt;
      p2 := pmin;
    end
    else
    begin
      p1 := pmin;
      p2 := par.pt;
    end;
    { See if outside limits, returning nearest endpoint or 0 if so, according
      to value of direction as outlined above. }
    if u < PtoU(w, p1, sc.ls[wv.ls[w]^.screen].gr.xaxismode) then
    begin
      if direction = -1 then
        UtoP := 0
      else
        UtoP := p1;
      exit;
    end
    else if u > PtoU(w, p2, sc.ls[wv.ls[w]^.screen].gr.xaxismode) then
    begin
      if direction = 1 then
        UtoP := 0
      else
        UtoP := p2;
      exit;
    end;
    { Find closest point according to direction criterion, using halving
      algorithm to close in on point iteratively. }
    while abs(p1 - p2) > 1 do
    begin
      p := (p1 + p2) div 2;
      up := PtoU(w, p, sc.ls[wv.ls[w]^.screen].gr.xaxismode);
      if u < up then
        p2 := p
      else if u > up then
        p1 := p
      else { Equal; can quit. }
      begin
        UtoP := p;
        exit;
      end;
    end;
    case direction of
    -1 :
      if u = PtoU(w, p2, sc.ls[wv.ls[w]^.screen].gr.xaxismode) then
        UtoP := p2
      else
        UtoP := p1;
```

```
    0 :
      if u - PtoU(w, p1, sc.ls[wv.ls[w]^.screen].gr.xaxismode) <
        PtoU(w, p2, sc.ls[wv.ls[w]^.screen].gr.xaxismode) - u then
        UtoP := p1
      else
        UtoP := p2;
    1 :
      if u = PtoU(w, p1, sc.ls[wv.ls[w]^.screen].gr.xaxismode) then
        UtoP := p1
      else
        UtoP := p2;
    end;
  end;
end;

function UtoX(u : real; scr : integer) : real;
{ Converts unit space x coordinate u to screen x coordinate. Note that value
  may fall outside of screen boundaries (which is why real value was used). }
begin
  with sc.ls[scr].gr do
    UtoX := xaxis.x1 + (u - u1lim) * (xaxis.x2 - 1 - xaxis.x1) /
      (u2lim - u1lim); { -1 is
    to ensure that points on righthand edge fall INSIDE screen limit. }
end;

procedure val_i(s : bufstring; var i : integer);
{ Integer version of clunky val() procedure which needs a dummy variable I
  never check (though is used here to set i to 0 for screw-ups). }
var
  error : integer;                { Error variable for val(). }
begin
  val(s, i, error);
  if error <> 0 then
    i := 0;
end;

procedure val_r(s : bufstring; var r : real);
{ Real version clunky val() procedure which needs a dummy variable I never
  check (though is used here to set r to 0 for screw-ups). }
var
  error : integer;                { Error variable for val(). }
begin
  val(s, r, error);
  if error <> 0 then
    r := 0;
end;

function VtoY(v : real; scr : integer) : real;
{ Converts unit space y coordinate v to screen y coordinate. Note that value
  may fall outside of screen boundaries (which is why real value was used). }
begin
  with sc.ls[scr].gr do
    VtoY := yaxis.y2 - 1 - (v - v1lim) * (yaxis.y2 - 1 - yaxis.y1) /
      (v2lim - v1lim); { -1
    is to ensure that points on bottom edge fall INSIDE screen limit. Note
    overall minus sign to expression, compared with UtoX. This is because we
    want (0, 0) in unit space to correspond to bottom left corner, not top
    left. }
end;

procedure waitkey;
{ Prints text message and waits for keypress. }
begin
  writeln(' ');
  writeln('Hit <return> to continue.');
  readln;
end;

function WaveInfo(w : integer; l : integer) : bufstring;
{ Prepare wave information in a string to be used by DrawWaveInfo. l is
  the length of the string to be returned (is padded with spaces). See
  help for format of output string. }
var
  i : integer; { General integer. }
  out : bufstring; { Growing output string. }
  s, s2 : bufstring; { General string. }
begin
  with wv.ls[w]^ do
  begin
    { Current wave. }
    if w = wv.cur then
      out := '*'
    else
      out := ' ';
    { Wave number. }
    str(w, s);
    out := out + s;
    { Invisible flag. }
    if wv.ls[w]^.on = 1 then
      out := out + ' '
    else
      out := out + 'i';
    { Saved flag. }
    if (wv.ls[w]^.parsaved = 1) and (wv.ls[w]^.datasaved = 1) then
    begin
      if wv.ls[w]^.savemode = 0 then
        out := out + 's'
      else
        out := out + 'n';
      end
    else
      out := out + ' ';
    { Screen number. }
    str(screen, s);
    out := out + s;
    { Mode. }
    if (addwaves.mode = 1) and (addwaves.w = w) then
      out := out + 'A'
    else if scan.mode > 0 then
      out := out + 'S'
    else if mon.w = w then
      out := out + 'M'
    else
      out := out + ' ';
    { Background subtraction flags: }
    if (par.dt = dt_ELE) and (par.ele.bs.mode = 1) then
    begin
      if _bs.bg = w then
        out := out + 'B'
      else
        out := out + 'b';
    end
    else
      out := out + ' ';
    { Selection flag. }
    if wv.ls[w]^.sel = 1 then
```

```
      out := out + '/'
    else
      out := out + ' ';
    { Need to fit two more items: label determined by wv.ls[]^.par.sh, and
      yscale (if in relative y axis mode). First determine amount of
      room left with yscale included, then add label, then yscale. }
    if sc.ls[wv.ls[w]^.screen].gr.yaxismode = YAXISMODE_RELATIVE then
    begin
      str(wv.ls[w]^.par.yscale:0:2, s2);
      s2 := ' ' + s2;
    end
    else
      s2 := '';
    maxwavenamelength := l - length(out) - length(s2);
    { Label determined by wv.ls[]^.par.sh. }
    case par.sh of
    wv_sh_CM : { Comment. }
      s := par.comment;
    wv_sh_DLY : { Time delay (electron wave only). }
      begin
        str(round(par.ele.dly / POWFS), s);
        s := s + ' fs';
      end;
    wv_sh_FN : { Filename. First figure out if path is same as current
      directory; if so, print only filename; otherwise, print full path.
      Then truncate filename to fit on screen, if necessary, in which
      case last two chars will be '..' to indicate truncation. }
{       begin
        i := FilenameStart(par.filename);
        if copy(par.filename, 1, i - 1) = dir then
          s := copy(par.filename, i, length(par.filename))
        else }
          s := par.fn;
{       end;    }
    end;
    if length(s) > maxwavenamelength then
      out := out + copy(s, 1, maxwavenamelength - 2) + '..'
    else
      out := out + s + copy(BLANKLINE, 1, maxwavenamelength - length(s));
      { Pad out to maximum length. }
    WaveInfo := out + s2; { Add in yscale and exit. }
  end;
end;

function WavesExist : integer;
{ Return 1 if wv.num > 0; 0 otherwise. }
begin
  if wv.num > 0 then
    WavesExist := 1
  else
    WavesExist := 0;
end;

function weight(x , a : real) : real;
{ Calculate weighting function for smoothing routines. }
begin
  weight := exp(-a * x * x);
end;

procedure writeengnotation(r : real);
begin
  write(r : PARENGDIGITS);
end;

procedure writeln_ctr(s : string; pos : integer);
{ Writes string centered on line at position pos. }
begin
  dec(pos, length(s) div 2);
  if pos > 0 then
    write(copy(BLANKLINE, 1, pos));
  writeln(s);
end;

procedure writevalue(r, p : real; dec, max : integer; u : string);
{ Writes value of dec decimals and maximum length max with proper SI
  prefix and unit string u. p is power of r; if p = 0, calculates p auto-
  matically. }
var
  s : bufstring; { Temporary string. }
begin
  if p = 0 then
    p := Power(r);
  str(r / p : 0 : dec, s);
  if length(s) > max then
    s := copy(s, 1, max);
  write(s, ' ', UnitPrefix(round(log10(p))), u);
end;

procedure writevalueunitless(r, p : real; dec, max : integer);
{ Writes value of dec decimals and maximum length max with power-of-ten
  attached. p is power of r. p = 0 is flag to calculate power automati-
  cally. }
var
  s : bufstring; { Temporary string. }
begin
  if p = 0 then
    p := Power(r);
  str(r / p : 0 : dec, s);
  if length(s) > max then
    s := copy(s, 1, max);
  if p <> 1 then
    write(s, ' (x', PowerOfTenPrefix(round(log10(p))), ')')
  else
    write(s);
end;

procedure ws_df;
{ Sets up default values in case where workspace read fails. }
var
  i, j : integer;
begin
  { Global vars. }
  for i := 1 to AD_MAX do
    ad.ls[i].gain := 1;
  ad.on := 1;
  allsaved := 1;
  auto.adv := 0;
  auto.bg := 1;
  auto.cor := 1;
  auto.fn := 1;
  auto.gen := 0;
  auto.num := 1;
  auto.off := 0;
  auto.rm := 0;
  auto.ser := AUTO_SER_DF;
  auto.sv := 0;
```

```
blankmin := 1;
blankmax := 100;
_bs.bg := 0;
_bs.dis := 0;
_bs.sts := 0;
_bs.sts_blank := 101;
_bs.sts_blank2 := 450;
_bs.sts_ch := 1;
_bs.sts_fac := 1;
_bs.sts_vert := 0.02;
_bs.sts_tog := 1;
dir := CD_DF;
discrim := 0.04;
dotradius := 1;
globalsavemode := 0;
mc.fn := '';
mc.num := 0;
mc.sens := 1;
mon.bins := 8;
mon.w := 0;
move.numavg := 10;
move.start := 0;
move.step := POWTS;
move.stop := 0;
move.wait := 0.2;
osc.ch := 1;
osc.scr := 1;
peaksecond := 0;       { set second derivative level to 0 }
peakthreshold := 0.1; { set to 10%}
printdir := CPD_DF;
setenergyconversionlaserev := 4;
setenergyconversionlength := 1.147;
setenergyconversiont0 := 247e-9;
t0.ac := 0;
t0.cc := 0;
ts.acc := TS_ACC_DF;
ts.step := TS_STEP_DF;
ts.vel := TS_VEL_DF;
ts.wait.int := TINT_DF;
ts.wait.sl := TSL_DF;
ts.wob.ampl := 0;
ts.wob.per := 0;
ts.wob.ph := 0;
tw.dly := -2e-12;
tw.pt := MCS_PT[MCS_PT_MIN];
tw.reprate := DF_REPRATE;
tw.shotsperscan := 500;
tw.timeperpt := 40e-9;
yoffsetrescale := 1;
{ Wave default vars. }
for i := dt_MIN to dt_MAX do
  with pardf[i] do
  begin
    alert := 0;
    comment := '';
    dt := i;
    fn := '';
    gen := 0;
    scan := 0;
    sh := wv_sh_FN;
    skip := 0;
    vstop := 0;
    yoffset := 0;
    yscale := 1;
  end;
with pardf[dt_COR] do
begin
  pt_gl := 50;
  pt := pt_gl;
  scan_gl := 1;
  timeperpt := POWTS / HALFSPEEDOFLIGHT;
  with cor do
  begin
    ch := 1;
    shotsperpt := 10;
    ts.start := 29975.5e-6;
    ts.step := TS_STEP_DF;
    ts.stop := 30024.5e-6;
    ts.t0 := 30000e-6;
    ts.wob.ampl := 0;
    ts.wob.per := 0;
    ts.wob.ph := 0;
  end;
end;
with pardf[dt_ELE] do
begin
  pt := MAXPOINTS;
  pt_gl := 0; { Doesn't matter. }
  scan_gl := 0; { Set for infinite scans. }
  timeperpt := MCS_TIMEPERPT[MCS_TIMEPERPT_MIN];
  with ele do
  begin
    bs.last := 0;
    bs.mode := 0;
    bs.tot := 0;
    cal.ev := 4;
    cal.len := setenergyconversionlength;
    cal.t0 := setenergyconversiont0;
    cal.quad := setenergyconversionquad;
    cal.quadoff := setenergyconversionquadoff;
    dly := 0;
    reprate := DF_REPRATE;
    shotsperscan := 10000;
    ts.pos := 30000e-6;
    ts.t0 := 30000e-6;
    ts.wob.ampl := 2.5;
    ts.wob.per := 10;
    ts.wob.ph := 0.525;
  end;
end;
with pardf[dt_POW] do
begin
  pt_gl := MAXPOINTS;
  pt := pt_gl;
  scan_gl := 1;
  timeperpt := 1;
  with pow do
  begin
    ch := 1;
    cal.int := 0;
    cal.sl := 1;
  end;
end;
with pardf[dt_MAS] do
begin
  pt := MAXPOINTS;
```

```
    pt_gl := 0; { Doesn't matter. }
    scan_gl := 1;
    timeperpt := 1e-6;
    with mas do
    begin
      cal.int := 0;
      cal.sl := 1;
      ch := 1;
      delay := 0;
      inv := 0;
      scantime := timeperpt * pt;
      vert := 0;
    end;
  end;
  { Write screen vars. }
  sc.cur := 1;
  sc.ls[sc.cur].gr.bdy := sc.bdy; { Copy max. limits. }
  sc_init(sc.cur); { Set default screen parameters. }
  sc.num := 0; { Must set this below sc.cur when calling sc_resize the
    first time, to establish xh's correctly. }
  sc_resize(sc.cur); { Establish screen coordinates for rest of graph. }
  sc.num := 1;
  ws_init;
end;

procedure ws_init;
{ Set a few variables to defaults whenever we change workspaces. }
const
  LF = #10; { Doesn't seem to work unless this is declared explicitly. }
var
  i : integer;
begin
  { Set these to 0 for first entering program. }
  if newworkspace = 1 then
  begin
    addwaves.mode := 0;
    for i := 1 to AD_MAX do
      ad.ls[i].on := 0;
    mon.tot := 0;
    newworkspace := 0;
    osc.mode := 0;
    osc.scr := 1;
    scanwave := 0;
    wv.num := 0;
  end;

  { Ensure all waves on actual screens. }
  for i := 1 to wv.num do
    with wv.ls[i]^ do
      if screen > sc.num then
        screen := sc.cur;
  { Ensure osc on actual screen. }
  if osc.scr > sc.num then
    osc.scr := sc.cur;
  Init_ts;
  graphicsmode;
  mc_rd(mc.fn);
  mc.s := '';
end;

procedure ws_rd(s : bufstring);
{ Reads ws file s from disk, which contains all critical system vars with
  the exception of wave data itself. If read fails (no file, or bad ver-
  sion), uses defaults in ws_df. }
var
  f : text; { File variable. }
  i, j : integer;
begin
  if FileExists(s) = false then
  begin
    if newworkspace = 1 then
    begin
      textmode;
      clrscr;
      textcolor(WHITE);
      writeln('Workspace file does not exist; using default values.',
        BELL);
      delay(1000);
      ws_df; { Set defaults. }
      exit;
    end
    else
    begin
      com_wr('Workspace file does not exist.', COLORHL);
      exit;
    end;
  end;
  Assign(f, s);
  Reset(f);
  readln(f, s);
  if s <> HEAD_WS then
  begin
    if newworkspace = 1 then
    begin
      textmode;
      clrscr;
      textcolor(WHITE);
      writeln('Workspace file format incompatible; using default values.'
        , BELL);
      delay(1000);
      ws_df; { Set defaults. }
      exit;
    end
    else
    begin
      com_wr('Workspace file format incompatible.', COLORHL);
      exit;
    end;
  end;
  { Read global vars. }
  for i := 1 to AD_MAX do
    readln(f, ad.ls[i].gain);
  readln(f, ad.on);
  readln(f, allsaved);
  readln(f, auto.adv);
  readln(f, auto.bg);
  readln(f, auto.cor);
  readln(f, auto.fn);
  readln(f, auto.gen);
  readln(f, auto.num);
  readln(f, auto.off);
  readln(f, auto.rm);
  readln(f, auto.ser);
  readln(f, auto.sv);
  readln(f, blankmin);
  readln(f, blankmax);
```

```
readln(f, _bs.bg);
readln(f, _bs.dis);
readln(f, _bs.sts);
readln(f, _bs.sts_blank);
readln(f, _bs.sts_blank2);
readln(f, _bs.sts_ch);
readln(f, _bs.sts_fac);
readln(f, _bs.sts_vert);
readln(f, _bs.sts_tog);
readln(f, debug);
readln(f, dir);
readln(f, discrim);
readln(f, dotradius);
readln(f, globalsavemode);
readln(f, mc.auto);
readln(f, mc.fn);
readln(f, mc.sens);
readln(f, mon.bins);
readln(f, mon.w);
readln(f, move.numavg);
readln(f, move.start);
readln(f, move.step);
readln(f, move.stop);
readln(f, move.wait);
readln(f, osc.ch);
readln(f, osc.scr);
readln(f, peaksecond);
readln(f, peakthreshold);
readln(f, printdir);
readln(f, setenergyconversionlaserev);
readln(f, setenergyconversionlength);
readln(f, setenergyconversiont0);
readln(f, setenergyconversionquad);
readln(f, setenergyconversionquadoff);
readln(f, t0.ac);
readln(f, t0.cc);
readln(f, ts.acc);
readln(f, ts.step);
readln(f, ts.vel);
readln(f, ts.wait.int);
readln(f, ts.wait.sl);
readln(f, ts.wob.ampl);
readln(f, ts.wob.per);
readln(f, ts.wob.ph);
readln(f, tw.dly);
readln(f, tw.pt);
readln(f, tw.reprate);
readln(f, tw.shotsperscan);
readln(f, tw.timeperpt);
readln(f, yoffsetrescale);
{ Read wave default vars. }
for i := dt_MIN to dt_MAX do
  with pardf[i] do
  begin
    readln(f, alert);
    readln(f, comment);
    readln(f, dt);
    readln(f, fn);
    readln(f, gen);
    readln(f, pt);
    readln(f, pt_gl);
    readln(f, scan);
    readln(f, scan_gl);
    readln(f, sh);
    readln(f, skip);
    readln(f, timeperpt);
    readln(f, vstop);
    readln(f, yoffset);
    readln(f, yscale);
  end;
with pardf[dt_COR].cor do
begin
  readln(f, ch);
  readln(f, shotsperpt);
  readln(f, ts.start);
  readln(f, ts.step);
  readln(f, ts.stop);
  readln(f, ts.t0);
  readln(f, ts.wob.ampl);
  readln(f, ts.wob.per);
  readln(f, ts.wob.ph);
end;
with pardf[dt_ELE].ele do
begin
  readln(f, bs.last);
  readln(f, bs.mode);
  readln(f, bs.tot);
  readln(f, cal.ev);
  readln(f, cal.len);
  readln(f, cal.t0);
  readln(f, cal.quad);
  readln(f, cal.quadoff);
  readln(f, dly);
  readln(f, reprate);
  readln(f, shotsperscan);
  readln(f, ts.pos);
  readln(f, ts.t0);
  readln(f, ts.wob.ampl);
  readln(f, ts.wob.per);
  readln(f, ts.wob.ph);
end;
with pardf[dt_POW].pow do
begin
  readln(f, ch);
  readln(f, cal.int);
  readln(f, cal.sl);
end;
with pardf[dt_MAS].mas do
begin
  readln(f, cal.int);
  readln(f, cal.sl);
  readln(f, ch);
  readln(f, delay);
  readln(f, inv);
  readln(f, scantime);
  readln(f, vert);
end;
{ Read screen vars. }
readln(f, sc.cur);
readln(f, sc.num);
for i := 1 to sc.num do begin
  with sc.ls[i] do begin
    readln(f, ti.on);
    readln(f, ti.s);
  end;
  with sc.ls[i].gr do begin
```

```
      readln(f, bdy[1].x);
      readln(f, bdy[1].y);
      readln(f, bdy[2].x);
      readln(f, bdy[2].y);
      readln(f, xh.mode);
      readln(f, cursoru);
      readln(f, cursorvisible);
      readln(f, u1lim);
      readln(f, u2lim);
      readln(f, v1lim);
      readln(f, v2lim);
      readln(f, xaxisdt);
      readln(f, xaxismode);
      readln(f, xfullmode);
      readln(f, xon);
      readln(f, yaxismode);
      readln(f, yfullmode);
      readln(f, yon);
      xh.which := 1;
      xh.u[1] := u1lim;
      xh.u[2] := u2lim;
      xh.v[1] := v1lim;
      xh.v[2] := v2lim;
      sc_resize(i); { Set up other screen variables. }
    end;
  end;
  { Done. }
  close(f);
  ws_init;
end;

procedure ws_sv(s : bufstring);
{ Saves system variables useful for startup each time in file s. }
var
  f : text; { File variable. }
  i, j : integer;
begin
  if s <> WS_FN_DF then { Only check for existence if not
    default file. }
    if FileExists(s) then
      if com_wr_yn('Workspace file exists. Overwrite') = 0 then
        exit;
  if FileOpenWrite(f, s) = false then
    exit;
  { Write version header. }
  writeln(f, HEAD_WS);
  { Write global vars. }
  for i := 1 to AD_MAX do
    writeln(f, ad.ls[i].gain);
  writeln(f, ad.on);
  writeln(f, allsaved);
  writeln(f, auto.adv);
  writeln(f, auto.bg);
  writeln(f, auto.cor);
  writeln(f, auto.fn);
  writeln(f, auto.gen);
  writeln(f, auto.num);
  writeln(f, auto.off);
  writeln(f, auto.rm);
  writeln(f, auto.ser);
  writeln(f, auto.sv);
  writeln(f, blankmin);
  writeln(f, blankmax);
  writeln(f, _bs.bg);
  writeln(f, _bs.dis);
  writeln(f, _bs.sts);
  writeln(f, _bs.sts_blank);
  writeln(f, _bs.sts_blank2);
  writeln(f, _bs.sts_ch);
  writeln(f, _bs.sts_fac);
  writeln(f, _bs.sts_vert);
  writeln(f, _bs.sts_tog);
  writeln(f, debug);
  writeln(f, dir);
  writeln(f, discrim);
  writeln(f, dotradius);
  writeln(f, globalsavemode);
  writeln(f, mc.auto);
  writeln(f, mc.fn);
  writeln(f, mc.sens);
  writeln(f, mon.bins);
  writeln(f, mon.w);
  writeln(f, move.numavg);
  writeln(f, move.start);
  writeln(f, move.step);
  writeln(f, move.stop);
  writeln(f, move.wait);
  writeln(f, osc.ch);
  writeln(f, osc.scr);
  writeln(f, peaksecond);
  writeln(f, peakthreshold);
  writeln(f, printdir);
  writeln(f, setenergyconversionlaserev);
  writeln(f, setenergyconversionlength);
  writeln(f, setenergyconversiont0);
  writeln(f, setenergyconversionquad);
  writeln(f, setenergyconversionquadoff);
  writeln(f, t0.ac);
  writeln(f, t0.cc);
  writeln(f, ts.acc);
  writeln(f, ts.step);
  writeln(f, ts.vel);
  writeln(f, ts.wait.int);
  writeln(f, ts.wait.sl);
  writeln(f, ts.wob.ampl);
  writeln(f, ts.wob.per);
  writeln(f, ts.wob.ph);
  writeln(f, tw.dly);
  writeln(f, tw.pt);
  writeln(f, tw.reprate);
  writeln(f, tw.shotsperscan);
  writeln(f, tw.timeperpt);
  writeln(f, yoffsetrescale);
  { Write wave default vars. }
  for i := dt_MIN to dt_MAX do
    with pardf[i] do
    begin
      writeln(f, alert);
      writeln(f, comment);
      writeln(f, dt);
      writeln(f, fn);
      writeln(f, gen);
      writeln(f, pt);
      writeln(f, pt_gl);
      writeln(f, scan);
      writeln(f, scan_gl);
```

```
      writeln(f, sh);
      writeln(f, skip);
      writeln(f, timeperpt);
      writeln(f, vstop);
      writeln(f, yoffset);
      writeln(f, yscale);
{       for j := USERMIN to USERMAXdt[i] do
        writeln(f, user[j]); }
    end;
  with pardf[dt_COR].cor do
  begin
    writeln(f, ch);
    writeln(f, shotsperpt);
    writeln(f, ts.start);
    writeln(f, ts.step);
    writeln(f, ts.stop);
    writeln(f, ts.t0);
    writeln(f, ts.wob.ampl);
    writeln(f, ts.wob.per);
    writeln(f, ts.wob.ph);
  end;
  with pardf[dt_ELE].ele do
  begin
    writeln(f, bs.last);
    writeln(f, bs.mode);
    writeln(f, bs.tot);
    writeln(f, cal.ev);
    writeln(f, cal.len);
    writeln(f, cal.t0);
    writeln(f, cal.quad);
    writeln(f, cal.quadoff);
    writeln(f, dly);
    writeln(f, reprate);
    writeln(f, shotsperscan);
    writeln(f, ts.pos);
    writeln(f, ts.t0);
    writeln(f, ts.wob.ampl);
    writeln(f, ts.wob.per);
    writeln(f, ts.wob.ph);
  end;
  with pardf[dt_POW].pow do
  begin
    writeln(f, ch);
    writeln(f, cal.int);
    writeln(f, cal.sl);
  end;
  with pardf[dt_MAS].mas do
  begin
    writeln(f, cal.int);
    writeln(f, cal.sl);
    writeln(f, ch);
    writeln(f, delay);
    writeln(f, inv);
    writeln(f, scantime);
    writeln(f, vert);
  end;
  { Write screen vars. }
  writeln(f, sc.cur);
  writeln(f, sc.num);
  for i := 1 to sc.num do begin
    with sc.ls[i] do begin
      writeln(f, ti.on);
      writeln(f, ti.s);
```

```
      end;
      with sc.ls[i].gr do begin
        writeln(f, bdy[1].x);
        writeln(f, bdy[1].y);
        writeln(f, bdy[2].x);
        writeln(f, bdy[2].y);
        writeln(f, xh.mode);
        writeln(f, cursoru);
        writeln(f, cursorvisible);
        writeln(f, u1lim);
        writeln(f, u2lim);
        writeln(f, v1lim);
        writeln(f, v2lim);
        writeln(f, xaxisdt);
        writeln(f, xaxismode);
        writeln(f, xfullmode);
        writeln(f, xon);
        writeln(f, yaxismode);
        writeln(f, yfullmode);
        writeln(f, yon);
      end;
    end;
    { Done. }
    close(f);
  end;

  procedure wv_sel(code : integer);
  { Examines command line for several levels of flags, depending on code:
    0: null, valid wave number, 'all' = all waves flagged, 'sel' = user
       flags waves with cursor. For all choices, subsequent processing
       must check wave[]^.sel to see which wave(s) were tagged.
    1: above options, plus 'enumeration list': several wave numbers listed
       explicitly in command. wv.ls[]^.sel holds list of waves indicated
       as for above.
    Sets wv.sel to 0 if no waves selected (if wv.num = 0, or if sel
    option returned no tags); otherwise, sets to 1. Leaves com.cur pointing
    to last recognized word (initial command, 'all', 'sel' or last enumera-
    tion list number.
  }
  var
    dummy : integer; { Dummy variable for val. }
    exitflag : boolean; { Flag for enumeration list. }
    i : integer; { wave counter. }
    s : bufstring; { Holds current word. }
  begin
    { Check to make sure there are waves. }
    if wv.num = 0 then
    begin
      wv.sel := 0;
      exit;
    end;
    { Set tags to 0. }
    for i := 1 to wv.num do
      wv.ls[i]^.sel := 0;
    if com.cur = com.num then
    { Null. }
    begin
      wv.ls[wv.cur]^.sel := 1;
      wv.sel := 1;
    end
    else
    begin
      inc(com.cur);
```

```
s := com.ls[com.cur];
val(s, i, dummy);
if (i >= 1) and (i <= wv.num) then
begin
  wv.ls[i]^.sel := 1;
  wv.sel := 1;
  { Read in more wave numbers if code allows us to. }
  if code = 1 then
  begin
    exitflag := false;
    while (com.cur < com.num) and (not exitflag) do
    begin
      inc(com.cur);
      val(com.ls[com.cur], i, dummy);
      if (i >= 1) and (i <= wv.num) then
        wv.ls[i]^.sel := 1
      else
      begin
        dec(com.cur); { Let calling routine figure out what meant. }
        exitflag := true;
      end;
    end;
  end;
end
else if s = 'all' then
begin
  for i := 1 to wv.num do
    wv.ls[i]^.sel := 1;
  wv.sel := 1;
end
else if s = 'sel' then
begin
  com_wr('Select waves: UP/DOWN = move, / = toggle, a = all, c = '
    + ' clear, t = toggle all, RETURN = execute, ESC = abort.',
    COLORMESS);
  exitflag := false;
  drawwavedata; { Initialize appearance of waves. }
  repeat
    Scan; { Take care of active waves. }
    if keypressed then
    begin
      case readkey of
      EXTENDED:
        case readkey of
          XARROWDOWN:
            if wv.cur < wv.num then
              inc(wv.cur)
            else
              wv.cur := 1;
          XARROWUP:
            if wv.cur > 1 then
              dec(wv.cur)
            else
              wv.cur := wv.num;
        end;
      CR: exitflag := true;
      ESC: { Abort selection: untag all waves. }
        begin
          for i := 1 to wv.num do
            wv.ls[i]^.sel := 0;
          exitflag := true;
        end;
      '/', '?': wv.ls[wv.cur]^.sel := 1 - wv.ls[wv.cur]^.sel;
          'a', 'A': { Select all. }
            for i := 1 to wv.num do
              wv.ls[i]^.sel := 1;
          'c', 'C': { Clear all. }
            for i := 1 to wv.num do
              wv.ls[i]^.sel := 0;
          't', 'T': { Toggle all. }
            for i := 1 to wv.num do
              wv.ls[i]^.sel := 1 - wv.ls[i]^.sel;
        end;
        DrawWaveData; { Update cursor, tags. }
      end;
    until exitflag;
    { See if any waves selected, returning proper value in wv.sel. }
    for i := 1 to wv.num do
      if wv.ls[i]^.sel = 1 then
      begin
        wv.sel := 1;
        exit;
      end;
    wv.sel := 0;
  end
  else
  begin { Unrecognized word; assume want current wave selected. }
    wv.ls[wv.cur]^.sel := 1;
    wv.sel := 1;
    dec(com.cur); { Leave word for calling routine to decipher. }
  end;
end;
{ DEBUG
s := 'wv_sel: ';
for i := 1 to wv.num do
  with wv.ls[i]^ do
    if sel = 1 then
      s := s + makestringint(i) + ', ';
com_wr(copy(s, 1, length(s) - 2), COLORDEBUG); }
end;

procedure wv_sel_off;
{ Turn off all sel tags on waves. }
var
  w : integer;
begin
  for w := 1 to wv.num do
    wv.ls[w]^.sel := 0;
  wv.sel := 0;
end;

procedure XFull(scr : integer);
{ Calculates largest u limits for currently visible waves. If all waves are
  invisible, then limits don't change (and they don't matter, either). }
var
  got_one : boolean; { Flag indicating valid wave found. }
  u1 : real; { Temporary unit variable. }
  u2 : real; { Temporary unit variable. }
  w : integer; { Wave index. }
begin
  with sc.ls[scr].gr do
  begin
    if xfullmode = 0 then
      exit;

    { Find first wave on screen which is visible, calculate its limits. }
```

```
got_one := false;
w := 1;
while (got_one = false) and (w <= wv.num) do
  with wv.ls[w]^ do
  begin
    if (on = 1) and (screen = scr) then
    begin
      if (xaxismode = XAXISMODE_CONVERT) and
        (xaxisdt = dt_ELE) then
      begin
        u2lim := PtoU(w, pmin, xaxismode);
        if par.pt > 0 then
          u1lim := PtoU(w, par.pt, xaxismode)
        else
          u1lim := u2lim;
      end
      else
      begin
        u1lim := PtoU(w, pmin, xaxismode);
        if par.pt > 0 then
          u2lim := PtoU(w, par.pt, xaxismode)
        else
          u2lim := u1lim;
      end;
      got_one := true;
    end;
    w := w + 1;
  end;
{ Now look for others, expand limits to include them all. }
if got_one then
  for w := w to wv.num do
    with wv.ls[w]^ do
    begin
      if (on = 1) and (screen = scr) then
      begin
        if (xaxismode = XAXISMODE_CONVERT) and (xaxisdt =
          dt_ELE) then
        begin
          u2 := PtoU(w, pmin, xaxismode);
          if par.pt > 0 then
            u1 := PtoU(w, par.pt, xaxismode)
          else
            u1 := u2;
        end
        else
        begin
          u1 := PtoU(w, pmin, xaxismode);
          if par.pt > 0 then
            u2 := PtoU(w, par.pt, xaxismode)
          else
            u2 := u1;
        end;
        if u1 < u1lim then
          u1lim := u1;
        if u2 > u2lim then
          u2lim := u2;
      end;
    end
else { No valid waves; use these defaults. }
begin
  u1lim := 0;
  u2lim := 1;
end;
  end;
end;

function XToTheY(x, y : real) : real;
{ Raises x to the y power. }
begin
  XToTheY := exp(ln(x) * y);
end;

function XtoU(x : longint; scr : integer) : real;
{ Calculates unit-space u coordinate, given screen coordinate x and
  screen number scr. }
begin
  with sc.ls[scr].gr do
    xtou := (x - plotarea.x1) * (u2lim - u1lim) / (plotarea.x2 -
      plotarea.x1 - 1) + u1lim;
end;

procedure YFull(scr : integer);
{ Displays all waves with full Y limits if yfullmode = 1. }
var
  got_one : boolean; { Flag indicating valid wave found. }
  v1 : real; { Generic unit variable. }
  v2 : real; { Generic unit variable. }
  w : integer; { Wave index. }
begin
  with sc.ls[scr].gr do
  begin
    if yfullmode = 0 then
      exit;
    { Find first valid wave, record its limits. }
    got_one := false;
    w := 1;
    while (got_one = false) and (w <= wv.num) do
    begin
      with wv.ls[w]^ do
        if (on = 1) and (screen = scr) then
        begin
          v1lim := vmin;
          v2lim := vmax;
          got_one := true;
        end;
      inc(w);
    end;
    { Now look for other waves and expand limits to include them all. }
    if got_one then
      for w := w to wv.num do
        with wv.ls[w]^ do
        begin
          if (on = 1) and (screen = scr) then
          begin
            v1 := vmin;
            v2 := vmax;
            if v1 < v1lim then
              v1lim := v1;
            if v2 > v2lim then
              v2lim := v2;
          end;
        end
    else
    begin { No waves -- use these limits. }
      v1lim := 0;
      v2lim := 1;
```

```
    end;
  end;
end;

procedure YScaleChange(r : real);
{ Change yscale of wave wv.cur by r. }
var
  rr : real; { Temporary value. }
begin
  with wv.ls[wv.cur]^ do
  begin
    rr := par.yscale + r;
    if rr = 0 then
      rr := rr + r; { Avoid actually getting to 0, which screws up yoffset-
        rescaling. }
    if yoffsetrescale = 1 then
      par.yoffset := par.yoffset * rr / par.yscale;
    wv.ls[wv.cur]^.par.yscale := rr;
  end;
  DrawWaveData;
end;

procedure YScaleChangeSign;
begin
  if wv.num = 0 then
    exit;
  wv.ls[wv.cur]^.par.yscale := -wv.ls[wv.cur]^.par.yscale;
  DrawWaveData;
end;

function YtoV(y : longint; scr : integer) : real;
{ Calculates unit-space u coordinate, given screen coordinate x and
  screen number scr. }
begin
  with sc.ls[scr].gr do
    YtoV := -(y - plotarea.y2 + 1) * (v2lim - v1lim) / (plotarea.y2 -
      plotarea.y1 - 1) + v1lim;
end;

procedure Zoom;
{ Only active in xh mode: Make screen boundaries equal to limits of
  xh. }
var
  r : real; { Temporary holder. }
begin
  with sc.ls[sc.cur].gr do
  begin
    if xh.mode = 0 then
      exit;
    with xh do
    begin
      xfullmode := 0;
      yfullmode := 0;
      u1lim := u[1];
      u2lim := u[2];
      if u1lim > u2lim then
      begin
        r := u1lim;
        u1lim := u2lim;
        u2lim := r;
      end;
      v1lim := v[1];
      v2lim := v[2];
      if v1lim > v2lim then
      begin
        r := v1lim;
        v1lim := v2lim;
        v2lim := r;
      end;
      { Make xh extend to full screen. }
      x[1] := plotarea.x1;
      x[2] := plotarea.x2 - 1;
      y[1] := plotarea.y1;
      y[2] := plotarea.y2 - 1;
      UpdateVitals;
      Update(sc.cur);
    end;
  end;
end;

end.
```

4.7. fpesvar.pas

```
unit fpesvar;
{ History of modifications (please add to BOTTOM of list!):

  Version 1: Begun 2jun94 BJG.

  Constant, type, and variable declarations for the program view. See view
  (at appropriate version) for more specific program modification notes.

  A note about the coordinate systems: there are three sets of coordinates to
  describe data positions in different "spaces." They are:

  (w, p) : wave and point space: w is a "wave" or data set; p is a "point" or
           bin number.
  (u, v) : unit space: u and v are axis coordinates in real units: for u, it is
           time (ns, us), energy (eV) or mass (amu); for v, it is either abso-
           lute intensity (counts) or relative intensity (arbitrary units).
           Which units are in use depends on the display mode: xaxismode_ and
           yaxismode keep track of these.
  (x, y) : screen space: x and y are screen coordinates used to plot graphics.

  These systems are always distinguished by using variable names incorporating
  the letters w and p, u and v, or x and y.
}

interface

uses
  crt, graph, Keys;

{ General purpose section***********************************************}
const
  ARRAYMAX = 10000; { Maximum reasonable array position for chararray,
    wordarray. }
  AUTO_SER_DF = 'file-'; { auto.ser default string. }
  BLANKLINE = '
';
    { 80 spaces. }
  COLORDEBUG = LIGHTGREEN; { Default debug message color. }
  COLORUSER = YELLOW; { Default user input color. }
```

```
  COLORMACRO = LIGHTMAGENTA; { Default macro color. }
  COLORMESS = LIGHTCYAN; { Default message color. }
  COLORHL = LIGHTRED; { Highlighting color (most routines also generate
    beep when using this color). }
  CD_DF = 'd:\fpes\data\'; { Default data directory. }
  CPD_DF = 'f:\fpes\print\'; { Default print directory. }
  DF_REPRATE = 500;
  ELECTRONMASS = 5.4863E-4; { Mass (amu) of electron - for energy conver-
    sion. }
  ELECTRONMASSKG = 9.10953E-31; { Electron rest mass in kilograms }
  ERR_DATAFORMAT = 'Error: Bad data format.';
  ERR_FILENOTFOUND = 'Error: file not found.';
  EVNM = 1239.842447; { eV.nm conversion factor between laser wavelength and
    energy. }
  FPES_EXT = 'par';
  HALFSPEEDOFLIGHT = 1.49896229e8; { Half the speed of light in m/s. }
  HEAD_WS = 'FPES WORKSPACE, REV. 1998.02.20';
  HEAD_WV = 'K - FPES WAVE FILE, REV. 20FEB98';
  HELP_FN = 'help.par';
  HELP_LINESPERPAGE = 24;
  JTOEV = 6.24146e18; { Joules to eV conversion factor }
  LABEL_END_CHAR = ':'; { Used to terminate label field in wv or ws
    files. }
  LARGE = 1e38; { Effective upper bound for range checking. }
  LARGEINT = 2147483647; { Effective longint upper bound. }
  LINESPERPAGE = 15; { for output of peaks }
  LN_2  = 0.693147181;                { ln(2). }
  LN_10 = 2.302585093;                { ln(10). }
  MAXPEAKS = 200;                     { ten screens of 20 peaks }
  MAXPOINTS = 1024; { max. # of data points in a wave. }
  MAXPOWER = 12; { Max. legal exponent in UnitPrefix. }
  MINPOWER = -18; { Minimum legal exponent in UnitPrefix. }
  POWFS = 1e-15; { Power for all fs vars. }
  POWTS = 1e-6; { Power for all ts vars. }
  SLOPEFACTOR = 2.842815689e-12; { Conversion factor (eV m^-2 s^2) for time
    to energy in PtoU. }
  SMALL = 1e-3; { Small number to add to rounding expressions to suppress
    roundoff error. }
  SQRTPI = 1.772453851; { Square root of Pi }
  SQRTLN_2 = 0.832554611; { Square root of the nat. log of 2 }
  SQRTLN_100 = 2.145966026; { Square root of the nat. log of 100 }
  TEKREPRATE = 100; { Max. rep. rate of Tektronix scope (Hz). }
  TEKSET_MAX = 30;
  TEK_YPTSPERDIV = 3276.8;
  TWOPI = 6.28318539717959; { 2 * pi. }
  VALDEC = 6; { Number of decimals for normal values printed. }
  VALDECFS = 1; { Number of decimals for fs values. }
  VALDECTS = 2; { Number of decimals for ts values. }
  VALMAX = VALDEC + 2; { Max. number of digits for normal values (counts
    '-' and '.') }
  VALMAXFS = 7; { Number of digits for fs values. }
  VALMAXTS = 8; { Number of digits for ts values. }
  VERSION = 'Version 6'; { Version. }
  WS_FN_DF = 'WS.PAR';
type
  bufstring = string[100]; { General-purpose buffer string. }
  chararray = array[1 .. ARRAYMAX] of char; { This definition, together
    with wordarray below, enable a pointer to reference any element of
    an arbitrary-sized array, when a pointer is declared as pointing to
    this array (much like c does for all pointer types). For instance,
    for p : ^chararray, can reference element i with p^[i]. }
  fft_array_type = array[1 .. 2 * MAXPOINTS] of real; { For use in fft
    and associated SmoothEnergy routines. }
  intbuf_type = array[1 .. MAXPOINTS] of integer; { Temp. array for data
    transfer during reads. }
  ts_wob_type = record { Software wob parameters to correct wobble
    in ts motion. Used by COR, ELE datatypes, also ts.wob main var. }
    ampl : real; { Amplitude. }
    per : real; { Period. }
    ph : real; { Phase. }
  end;
  wordarray = array[1 .. ARRAYMAX] of word; { See chararray above. }
  xy_type = record  { Screen-space position pair. }
    x, y : integer;
  end;
  { Note: the following definitions must be out of order, since they use
    other definitions which would normally follow after them alphabeti-
    cally. }
  bdy_type = array[1..2] of xy_type; { Pair of coodinates describing
    graphics boundary. }
  tx_type = record
    bdy : array[1..2] of xy_type; { Screen boundaries. }
    buf : ^chararray; { Pointer to chunk of memory holding text data. }
    col : ^wordarray; { Pointer to color list (1 for each line). }
    cur : xy_type; { Current x and y char position. }
    num : xy_type; { Max. number of characters in x and y directions. }
  end;
  tx_type_p = ^tx_type;
var
  allsaved : integer; { Flag for whether all waves have been saved. }
  auto : record { Auto-filename variables. }
    adv : real; { Number of s to advance stage when current ele scan
      is done. Restarts new scan automatically. If 0, mode is off. }
    bg : integer; { Flag to scale bg and foreground waves. }
    cor : integer; { Flag to scale and calculate fwhm for cor waves. }
    fn : integer; { Flag to generate filenames with new waves. }
    gen : integer; { Flag to generate new wave when scan done. }
    num : integer; { File number counter (for fn = 1 mode). }
    off : integer; { Flag to turn off scans if all electron waves are
      off. }
    rm : integer; { Flag to remove waves if full. }
    ser : bufstring; { Series name for files (for fn = 1 mode). }
    sv : integer; { Flag to save automatically when scan done. }
  end;
  blankmin, blankmax : integer; { Current point range to blank using
    Blank-Data routine. }
  _bs : record { Background subtraction mode parameters. }
    bg, fg : integer; { Background and foreground waves. }
    dis : integer; { Flag for bs display mode. }
    sts : integer; { Flag for sts (shot-to-shot) bs mode. }
    sts_blank : integer; { Number of initial bins to blank to elimi-
      nate noise spike. }
    sts_blank2 : integer; { Number of final bins to blank. }
    sts_ch : integer; { Channel number on TEK to use for bg scans. }
    sts_fac : real; { Conversion factor for scaling TEK and MCS
      waves. }
    sts_tog : integer; { Flag indicating whether to use toggle mode or
      regular mode for data accumulation (regular usually means will
      connect chopper to the inhibit BNC, so that background scans are
      not accumulated). }
    sts_vert : real; { Vertical scale used on TEK. }
  end;
  calib : record { Temporary holder of parameters for calibrating an
    electron or ion spectrum. }
    c : array[1..2] of real; { Converted values (energy or mass). }
    n : array[1..2] of real; { Normal values (time). }
```

```
end;
debug : integer; { Flag to disable graphics screen and print debugging
  messages. }
dev_rd : integer; { Flag to disable device reading, so can use for
 analysis on PC not connected to devices. }
dir : bufstring; { Current directory label. }
discrim : real; { Discrimination level used for electrons (MCS and
  TEK). }
exitflag : boolean; { Flag to end program. }
filter : array[1 .. MAXPOINTS div 2 + 1] of real; { Filter function
  used by SmoothEnergy. }
globalsavemode : integer;  { save wave mult by yscale or not. }
info : record { Vars calculated with info functions. }
  area : real;
  ctr : real;
  edgel : real;
  edger : real;
  fwhm : real;
end;
macro_override : integer; { macro override flag }
mon : record { Parameters for mon(itor) command. }
  bins : integer; { Number of bins to add together for tweak wave. }
  tot : real; { Total counts. }
  w : integer; { Tweak wave. }
end;
newworkspace : integer; { Flag to indicate new workspace (1) or not
  (0). }
osc : record { Virtual oscilloscope. }
  bit : pointer; { Storage for graphics under line. }
  ch : integer; { Channel to display. }
  mode : integer; { Flag indicating if on. }
  scr : integer; { Screen displayed on. }
  y : integer; { Screen-space y coordinate. }
end;
peakthreshold : real;       { numerical threshold [0..1] for peak finding }
peaksecond : real;          { second derivative value for peak finding }
printdir : bufstring;       { Current directory prints are sent to. }
setenergyconversionlaserev : real; { User-specified values to set all }
setenergyconversionlength : real; { energy waves to (see SetEnergyCon- }
setenergyconversiont0 : real;
setenergyconversionquad : real;
setenergyconversionquadoff : real;
t0 : record { Record of ac and cc t0 positions -- defaults. }
  ac : real;
  cc : real;
end;
tekset : record
  num : integer;
  s : array[1 .. TEKSET_MAX] of bufstring;
end;
tw : record { Parameters for tw (tweak) command. }
  dly : real; { Delay position (relative to pardf[dt_ELE].ele.t0)
    for tweaking. }
  pt : integer; { Number of points to read. }
  reprate : real;
  shotsperscan : integer; { Records per scan. }
  starttime, steptime : longint; { Internal timing vars. }
  timeperpt : real; { Bin size. }
end;
ynaesc_response : integer; { Holds last ResponseFull/ResponseYesNo result. }
yoffsetrescale : integer; { Flag indicating whether to rescale wave[]^.par
  .yoffset when wave[]^.par.yscale is changed. }
```

```
{ col section (colors)*************************************************}
const
  COLORMAX = 15; { Max. colors. }
  COLORNAME : array [1 .. COLORMAX] of bufstring = ('blue', 'green',
    'cyan', 'red', 'magenta', 'brown', 'lightgray', 'darkgray',
    'lightblue', 'lightgreen', 'lightcyan', 'lightred', 'lightmagenta',
    'yellow', 'white'); { Names of colors in order as shown in Program-
    mer's reference 7.0, p. 29. For use with user specification of wave
    colors. }
  { Following must be out of order since definition uses COLORMAX. }
  COLOR : array [1 .. COLORMAX] of word = (LightRed, Yellow, LightGreen,
    LightCyan, LightBlue, LightMagenta, White, Red, Brown, Green, Cyan,
    Blue, Magenta, LightGray, DarkGray); { List of wave colors
    (in order of most to least visible. }
var
  anticolor : array[1 .. COLORMAX] of integer; { Color-to-code index. }

{ com section (command line)******************************************}
const
  com_ls_MAX = 30; { Maximum number of words in com.ls[]. }
  com_wr_MOVINGSTAGE = 'Moving stage. Press any key when ready.';
var
  com : record
    ls : array[1 .. com_ls_MAX] of bufstring; { List of words typed by
      user. }
    cur : integer; { Current word being looked at. }
    num : integer; { Number of words in ls[]. }
    old : string; { Last string typed. }
    sv : string; { User input save string. }
    tx : tx_type; { Text screen variables. }
    ystart : integer; { Starting line of input. }
  end;

{ gr section (graphics screen)****************************************}
const
  CURSORLENGTH = 30;                { Length of cursor in pixels. }
  MAXDOTSIZE = 10;                  { largest sensible dot size for screen }
  MAXXDIGITS = 6; { Max. # of digits for x-axis numbers. }
  MAXYDIGITS = 6; { Max. # of digits for y-axis numbers. }
  XAXISMODEMIN = 0;
  XAXISMODEMAX = 2;
  XAXISMODE_POINTS = 0; { xaxismode states: display points only. }
  XAXISMODE_NORMAL = 1; { *: display normal x unit (time or distance, depen-
    ding on dt. }
  XAXISMODE_CONVERT = 2; { *: display converted x unit (energy, mass or
    time, depending on dt. }
  YAXISMODE_ABSOLUTE = 0;
  YAXISMODE_RELATIVE = 1;
type
  box = record
    xmax, ymax : integer; { Max. x and y character units. }
    x1, x2, y1, y2 : integer; { Screen coordinates defining a box. }
  end;
  fillPatternType = array [1..8] of byte; { ??? }
  gr_type = record
    bdy : bdy_type; { Max. screen boundaries. }
    cursorp : integer;          { Point index of cursor. }
    cursoru : real;             { Unit space x coordinate of cursor. }
    cursorvisible : integer;    { Flag indicating whether cursor is visi-
                                  ble (1) or not (0). }
    cursorx : integer;          { Screen x coordinate of cursor. }
    cursory1 : integer;         { Screen starting y coordinate of cursor. }
    cursory2 : integer;         { Screen ending y coordinate of cursor. }
```

```
    maxxnums : integer;          { Max. number of numbers to print along x
                                   axis. }
    maxynums : integer;
    plotarea : box;              { Screen limits for plotting area. }
    u1lim : real;                { Left limit of displayed data (user space). }
    u1norm : real;               { Left normalization position. Used in
                                   normalization = INTEGRAL, SELECTED. }
    u1num : real;                { Leftmost number displayed on x axis. }
    u2lim : real;                { Right limit of data. }
    u2norm : real;               { Right normalization position. Used in
                                   normalization = INTEGRAL. }
    u2num : real;                { Rightmost number displayed on x axis. }
    ustep : real;                { Step size for x axis numbers. }
    v1lim : real;                { Bottom limit of data. }
    v1num : real;                { Bottommost number displayed on y axis. }
    v2lim : real;                { Top limit of displayed data. }
    v2num : real;                { Topmost number displayed on y axis. }
    vstep : real;                { Step size for y axis numbers. }
    xaxis : box; { Screen limits for x axis. }
    xaxisdt : integer; { If xaxismode <> XAXISMODE_POINTS, then this
      variable tells us which dt can be displayed. }
    xaxismode : integer; { X axis display format. }
    xdecimals : integer; { Number of decimals to print on x axis. Depends on
      dt. }
    xfullmode : integer; { Full-scale flag for x axis. }
    xh : record { Crosshairs. }
      bitmap : record
        x, y : array[1 .. 3] of pointer; { Pointer to storage areas for
          graphics under crosshairs. Defined in InitGraphics. }
      end;
      mode : integer; { Flag indicating whether on or off. }
      u, v : array[1 .. 3] of real; { Unit-space locations of the x and y
        cursor hairs (1 = first, 2 = second, 3 = center. }
      which : integer; { Flag indicating which crosshairs the arrows manipu-
        late: 1 = x[1], y[1]; 2 = x[2], y[2]. }
      x, y : array[1 .. 3] of longint; { Screen-space locations. }
    end;
    xlabel : box; { Screen limits for x label. }
    xlabelstring : bufstring;  { X axis label (depends on scantype). }
    xnumbers : box; { Screen limits for x numbers. }
    xon : integer; { Flag to display x axis. }
    xpower : real; { Power of 1000 to divide x axis labels by for display. }
    xunits : bufstring; { Units for x axis (depends on scantype). }
    yaxis : box; { Screen limits for y axis. }
    yaxismode : integer; { Display mode for y axis. }
    ydecimals : integer;
    yfullmode : integer; { Full scale flag for y axis. }
    ylabel : box; { Screen limits for y label. }
    ylabelstring : bufstring;
    yon : integer; { Flag to display y axis. }
    ypower : real; { Power of 1000 to divide y axis labels by for display. }
    yunits : bufstring;
    ynumbers : box; { Screen limits for y numbers. }
  end;
var
  bitmap : pointer;              { Pointer to storage area for graphics
                                   under cursor. Defined in InitGraphics. }
  cursorinfo : box;              { Screen limits in which information about
                                   a data point is plotted. }
  dotradius : integer; { radius of dots on screen }
  graphmode : integer;           { Variable needed by graphics routines. }
  maxwavenamelength : integer; { Max. length of wavenames displayed. }
  messagebox : box;              { Screen limits for graphics messages. }
  textsize : integer;            { Size of graphics character in pixels (X
                                   and Y are the same). Used to lay out
                                   screen correctly regardless of graphics
                                   mode. }
  wavedata : box; { Screen limits for wave information. }

{ mc section (mass calculator)*****************************************}
const
  MC_LABEL_MAX = 10; { Number of chars in label. }
  MC_MAX = 20; { Number of elements. }
  MC_LINESPERSCREEN = 24;
type
  mc_label = string[MC_LABEL_MAX];
var
  mc : record { Mass calculator parameters. }
    auto : integer; { Flag indicating if on. }
    fn : bufstring; { Name of current mc file (so can ws can reload). }
    ls : array[1 .. MC_MAX] of record
      m : integer; { Mass. }
      min, max : integer; { Min and max. number of times can be used. }
      num : integer; { Temporary storage of mass combination. }
      s : mc_label; { Element label. }
    end;
    num : integer; { Number of elements in ls[] array. }
    s : bufstring; { String printed to cursorinfo box containing
      current combination(s). }
    sens : real; { Sensitivity of calculator (# of mass units). }
  end;

{ sc section (screen)**************************************************}
const
  sc_MAX = 12;
  sc_mode_GR = 0;
  sc_mode_TX = 1;
  sc_mode_TX_OVR = 2; { Text override mode -- to force screen to stay
    in text mode even if graphics command issued. This is to allow
    active waves to update, but not redraw, while doing something in
    text mode (such as editing a wave). }
type
  sc_type = record
    sel : integer; { Flag indicating if selected. }
    ti : record { Screen title block. }
      bdy : bdy_type; { Screen limits. }
      on : integer; { Flag to display. }
      s : bufstring; { Title. }
    end;
    { Screen variables. Two types are possible, indicated by mode. This
      construction is called a variable record: gr and tx share the same
      space in memory, and therefore only one is valid at a time. }
    case mode : integer of
      sc_mode_GR : (gr : gr_type);
      sc_mode_TX : (tx : tx_type);
  end;
var
  sc : record
    bdy : bdy_type; { Total screen area limits. }
    cur : integer; { Current screen. }
    ls : array[1 .. sc_MAX] of sc_type; { Screen variables. }
    mode : integer; { Overall screen mode: sc_mode_GR (normally), or
      sc_mode_TX (for editing waves, etc.) }
    num : integer; { Number of active screens. }
    sel : integer; { Flag indicating if any screens were selected. }
  end;
```

```
{ scan section (overall scanning)************************************}
const
  AD_MAX = 8;
  SCAN_MODE_NEW = 5;
  TIMEMAX = 100 * 60 * 60 * 24; { Maximum value returned from time(). }
var
  addwaves : record
    mode : integer; { Flag indicating if on. }
    w : integer; { Wave holding added result. }
  end;
  ad : record { A/D variables. }
    ls : array[1 .. AD_MAX] of record
      gain : integer; { Gain factor = 1, 2, 4, 8. }
      on : integer; { Whether channel is active. }
      result : integer; { Storage of A/D result. }
    end;
    on : integer; { Overall flag to read or not. }
  end;
  ibbuf : array[1 .. 2 * MAXPOINTS] of char; { Data transfer array for
    GPIB card. }
  intbuf : ^intbuf_type; { Pointer to ibbuf, but integer format. }
  magic : integer; { Gpib variable for magic controller. }
  mcs : record { SRS Multichannel Scalar variables. These values are
    kept here so we only need to talk to Scalar when parameters have
    changed. }
    addr : integer; { Gpib address. }
    new : integer; { Flag indicating if MCS needs to be initialized. }
    pt : integer; { Number of points. }
    shotsperscan : integer; { Records per scan. }
    timeperpt : real; { Bin width (s). }
  end;
  rdbuf : bufstring; { rdbuf : array[1..$1000] of char; see ibbuf. }
  reprate : integer; { Reprate of laser (Hz). }
  scanwave : integer; { Current wave being scanned. }
  starttime : longint; { The time the escan starts. }
  tek : integer; { Gpib variable for Tektronix scope. }

{ ts section (translation stage)************************************}
const
  ACKDELAY = 250; { Number of ms to wait after sending ACK code. }
  TINT_DF = 20; { Empirical minimum time for stage delay. }
  TS_ACC_DF = 200000; { Default acceleration (um s^-2). }
  TS_LOOKUP_MAX = 32; { Maximum entry in lookup table. Should be multi-
    ple of 2 for most efficient lookup. }
  TS_PHASE_MAX = 1; { Range of wob phase parameter. }
  TS_PHASE_MIN = 0; { " }
  TS_RES_STEP = 4e-5; { Resolution-stepsize product of ts, e.g. res. 40
    = 1 um, res. 800 = 0.05 um, etc. }
  TS_STEP_DF = 1e-6; { Standard step size. }
  TS_STEP_MIN = TS_RES_STEP / 1024; { Min. stepsize of ts, at res =
    1024. }
  TS_VEL_DF = 4000; { Default velocity (um/s). }
  TS_VEL_MAX = 30000; { Maximum velocity (um/s). }
  TSL_DF = 100000000 div TS_VEL_DF; { Slope for timing in 1/100 s per
    m. }
var
  com2 : text; { Text file pointer for translation stage. }
  move : record { Stage variables. }
    numavg, start, step, stop, wait : real;
  end;
  ts : record
    acc : longint; { Current acceleration (sent at start of program, and
      whenever stage is initialized). }
    lookup : array[0 .. TS_LOOKUP_MAX] of real; { Lookup table for
      wob function. }
    pos : real; { Current stage position. }
    step : real; { Current step size of stage. }
    vel : longint; { Current velocity (sent at start of program, and
      whenever stage is initialized). }
    wait : record { Waiting parameters for stage motion. }
      int, sl : longint; { Intercept and slope. }
    end;
    wob : ts_wob_type; { Wobble wob parameters. }
  end;

{ tx section (text screen)*****************************************}
{ All definitions in general section, due to ordering of other struc-
  tures which also use them. }

{ wv section (waves)***********************************************}
const
  USERMIN = 16; { Min. element of wave[]^.par.user[]. }
  USERMAX = USERMIN + 15; { Max. element of wave[]^.par.user[]. }
  CHANGEPARXNAME = 1; { X position of names in changepar procedure. }
  CHANGEPARXVALUE = 40; { X position of values " }
  CHANGEPARYSTART = 2; { Y position of start of parameters " }
  CHANGEPARYUSER = CHANGEPARYSTART + USERMIN; { Y position of user
    parameters " }
  dt_COR = 1;           { Values of par.dt. }
  dt_ELE = 2;
  dt_POW = 3;
  dt_MAS = 4;
  dt_MIN = 1;
  dt_MAX = 4;
  dt_NAME : array[dt_MIN .. dt_MAX] of string = ('cor', 'ele', 'pow',
    'mas');
  MAS_CH_MAX = 4; { Max. channel # for par.mas.ch. }
  MAXdt = 4; { Number of kinds of data types. }
  MAXNUMSCANS = 10000; { Arbitrary upper limit to NUMSCANS. }
  MAXSHOTSPERPT = 32000; { Arbitrary upper limit to PAR_C_SHOTSPERPT. }
  MAXSHOTSPERSCAN = 32000; { Max. shots per scan on MCS. }
  MAXWAVES = 18; { max. # of waves in memory at a time. }
  MCS_PT_MIN = 1;
  MCS_PT_MAX = 1;
  MCS_PT : array[MCS_PT_MIN .. MCS_PT_MAX] of integer = (1024);
    { Codes understood by the MCS corresponding to number of points. }
  MCS_TIMEPERPT_MIN = 0;
  MCS_TIMEPERPT_MAX = 10;
  MCS_TIMEPERPT : array [MCS_TIMEPERPT_MIN .. MCS_TIMEPERPT_MAX] of
    real = (5e-9, 40e-9, 80e-9, 160e-9, 320e-9, 640e-9, 1.28e-6,
    2.56e-6, 5.12e-6, 10.24e-6, 20.48e-6); { Codes understood by MCS
    corresponding to time per point. }
  { wv.ls[]^.par.user indices for different datatypes: }
  { For all datatypes: }
  PAR_ALERT = 1;
  PAR_COMMENT = 2;
  PAR_DT = 3;
  PAR_FILENAME = 4;
  PAR_GEN = 5;
  PAR_PT = 6;
  PAR_PT_GL = 7;
  PAR_SCAN = 8;
  PAR_SCAN_GL = 9;
  PAR_SH = 10;
  PAR_SKIP = 11;
```

```
PAR_TIMEPERPT = 12;
PAR_VSTOP = 13;
PAR_YOFFSET = 14;
PAR_YSCALE = 15;
{ For dt_COR: }
PAR_C_CHANNEL = USERMIN;
PAR_C_SHOTSPERPT = USERMIN + 1;
PAR_C_STAGESTART = USERMIN + 2;
PAR_C_STAGESTEP = USERMIN + 3;
PAR_C_STAGESTOP = USERMIN + 4;
PAR_C_STAGET0 = USERMIN + 5;
PAR_C_STAGEwobAMPL = USERMIN + 6;
PAR_C_STAGEwobPER = USERMIN + 7;
PAR_C_STAGEwobPH = USERMIN + 8;
{ For dt_ELE: }
PAR_E_BS_LAST = USERMIN;
PAR_E_BS_MODE = USERMIN + 1;
PAR_E_BS_TOT = USERMIN + 2;
PAR_E_CALEV = USERMIN + 3;
PAR_E_CALLENGTH = USERMIN + 4;
PAR_E_CALT0 = USERMIN + 5;
PAR_E_REPRATE = USERMIN + 6;
PAR_E_SHOTSPERSCAN = USERMIN + 7;
PAR_E_STAGEPOS = USERMIN + 8;
PAR_E_STAGET0 = USERMIN + 9;
PAR_E_STAGEwobAMPL = USERMIN + 10;
PAR_E_STAGEwobPER = USERMIN + 11;
PAR_E_STAGEwobPH = USERMIN + 12;
PAR_E_TIMEFPES = USERMIN + 13;
PAR_E_CALQUAD = USERMIN + 14;
PAR_E_CALQUADOFF = USERMIN + 15;
{ For dt_POW: }
PAR_P_CHANNEL = USERMIN;
PAR_P_CALINT = USERMIN + 1;
PAR_P_CALSLOPE = USERMIN + 2;
{ For dt_MAS: }
PAR_M_CALINT = USERMIN;
PAR_M_CALSLOPE = USERMIN + 1;
PAR_M_CHANNEL = USERMIN + 2;
PAR_M_DELAY = USERMIN + 3;
PAR_M_INV = USERMIN + 4;
PAR_M_SCANTIME = USERMIN + 5;
PAR_M_VERT = USERMIN + 6;
{ Labels for use on screen and in wave files. }
PARLABEL : array[1 .. USERMIN - 1] of string =
( 'Alert (alert)',
  'Comment (cm)',
  'Datatype',
  'Filename (fn)',
  'Generate',
  'Number of points',
  'Number of points, goal',
  'Number of scans',
  'Number of scans, goal',
  'Show (sh)',
  'Skip (skip)',
  'Time per point',
  'V stop',
  'Y offset',
  'Y scale' );
PARLABELDT : array[DT_MIN .. DT_MAX, USERMIN .. USERMAX] of string =
( ( 'A/D channel',
    'Shots per point',
```

```
    'Translation stage start',
    'Translation stage step',
    'Translation stage stop',
    'Translation stage zero of time',
    'Translation stage wobble amplitude',
    'Translation stage wobble period',
    'Translation stage wobble phase', '', '', '', '', '', '', '' ),
  ( 'Background wave last scan total',
    'Background subtraction mode',
    'Background wave all scans total',
    'Calibration maximum energy',
    'Calibration length',
    'Calibration time offset',
    'Repetition rate',
    'Shots per scan',
    'Translation stage position',
    'Translation stage time zero (t0)',
    'Translation stage wobble amplitude',
    'Translation stage wobble period',
    'Translation stage wobble phase',
    'Time delay of FPES (dly)',
    'Calibration quad factor',
    'Calibration quad offset factor' ),
  ( 'A/D channel',
    'Calibration intercept',
    'Calibration slope', '', '', '', '', '', '', '', '', '', '', '', '', ''),
  ( 'Calibration intercept',
    'Calibration slope',
    'Channel',
    'Delay',
    'Invert',
    'Scan time',
    'Vertical scale', '', '', '', '', '', '', '', '', '' ) );

PARDIGITS = 3; { Number of digits to display in parameters. }
PARDECIMALS = 5; { Number of decimals " }
PARENGDIGITS = 10; { Number of decimals for engineering notation (4-digit
  mantissa, 2-digit exponent, '-', '.', 'E', '-'). }
PARYSTART = 2; { For change parameter procedure. }
PARYSTOP = 25; { " }
STAGEMIN = 0; { Minimum translation stage position (m). }
STAGEMAX = 0.20; { Maximum translation stage position (m). }
TEK_PTPERDIV = 50;
TEK_TIMEPERPT_MIN = 1;
TEK_TIMEPERPT_MAX = 32;
TEK_TIMEPERPT : array[TEK_TIMEPERPT_MIN .. TEK_TIMEPERPT_MAX] of
  real = (1e-11, 2e-11, 4e-11, 1e-10, 2e-10, 4e-10, 1e-9, 2e-9, 4e-9,
  1e-8, 2e-8, 4e-8, 1e-7, 2e-7, 4e-7, 1e-6, 2e-6, 4e-6, 1e-5, 2e-5,
  4e-5, 1e-4, 2e-4, 4e-4, 1e-3, 2e-3, 4e-3, 1e-2, 2e-2, 4e-2, 1e-1,
  2e-1); { s. }
TEK_VPERDIV_MIN = 1;
TEK_VPERDIV_MAX = 13;
TEK_VPERDIV : array[TEK_VPERDIV_MIN .. TEK_VPERDIV_MAX] of real =
  (1e-3, 2e-3, 5e-3, 1e-2, 2e-2, 5e-2, 1e-1, 2e-1, 5e-1, 1, 2, 5,
  10); { V/div. }
USERMAXDT : array[1 .. dt_MAX] of integer = (USERMIN + 8,
  USERMIN + 15, USERMIN + 2, USERMIN + 6); { Max. user parameters for
  each dt. }
wv_sh_CM = 1; { Comment. }
wv_sh_DLY = 2; { Delay (dt_ELE only). }
wv_sh_FN = 3; { Filename. }
wv_sh_MAX = 3;
wv_sh_NAME : array[1 .. wv_sh_MAX] of string[3] = ('cm', 'dly', 'fn');
```

```
    { Names of wv_sh types for user interaction. }
  XUNITTYPE : array[dt_MIN .. dt_MAX, XAXISMODEMIN .. XAXISMODEMAX
    ] of string = ((‘’, ‘m’, ‘s’), (‘’, ‘s’, ‘eV’), (‘’, ‘s’, ‘s’),
    (‘’, ‘s’, ‘Da’));
type
  data_type = array[1 .. MAXPOINTS] of real;
  par_type = record
    { All datatypes. }
    alert : integer; { Flag to let user start new scan manually. }
    comment : bufstring;
    { dt: see case statement below. }
    fn : bufstring;
    gen : integer; { Flag to generate new wave when scans completed. }
    pt : integer; { Number of points in wave so far. }
    pt_gl : integer; { Goal number of points (for cor, pow scans) --
      scanning stops when pt reaches this number. }
    scan : integer; { Number of completed scans. }
    scan_gl : integer; { Goal number of scans -- scanning stops when
      scan reaches this number. }
    sh : integer; { Show code: indicates which variable is displayed in
      waveinfo box (wv_sh_FN: filename; wv_sh_DLY: delay (electron scan
      only); wv_sh_CM: comment). }
    skip : integer; { Number of cycles to skip between reads. }
    timeperpt : real;
    vstop : real; { Alternate way of ending scan: when vmax exceeds
      vstop. }
    yoffset : real;
    yscale : real;
    { Variable part. }
  case dt : integer of
    dt_COR :
    ( cor : record
        ch : integer;
        shotsperpt : integer;
        ts : record
          wob : ts_wob_type;
          start : real;
          step : real;
          stop : real;
          t0 : real;
        end;
      end; );
    dt_ELE:
    ( ele : record
        bs : record { Background subtraction variables. }
          last : real; { Last scan's bg counts -- for redo command. }
          mode : integer; { Enable/disable flag. }
          tot : real; { Total bg counts. }
        end;
        cal : record { Energy calibration parameters. }
          ev : real;
          len : real;
          t0 : real;
          quad : real;
          quadoff : real;
        end;
        dly : real; { Pump-probe delay. }
        reprate : real;
        shotsperscan : integer;
        ts : record
          wob : ts_wob_type;
          pos : real;
          t0 : real;
        end;
      end; );
    dt_POW:
    ( pow : record
        ch : integer;
        cal : record
          int : real;
          sl : real;
        end;
      end; );
    dt_MAS:
    ( mas : record
        cal : record
          int : real;
          sl : real;
        end;
        ch : integer;
        delay : real;
        inv : integer;
        scantime : real;
        vert : real;
      end; );
  end;
  par_type_ptr = ^par_type;
  wave_type = record
    col : word; { Wave color. }
    data : data_type; { Data for waves. }
    datasaved : integer; { Flag indicating whether data information has been
      saved. }
    lines : integer; { Flag indicating lines (1) or dots (0) are displayed. }
    mass1 : real; { Storage of first mass point during mass calibration;
      also a flag (= 0) that first mass point has not been entered yet. }
    on : integer; { Flag to display (1) wave or not (0). }
    par : par_type; { Parameters visible to user; see par_type. }
    parsaved : integer; { Flag indicating if par variables have been
      changed. }
    pmin : integer; { Minimum physically meaningful point when x axis units
      are energy or mass. }
    savemode : integer; { Indicates whether saved data is multiplied by par.
      yscale or not. }
    screen : integer; { Screen wave displays on. }
    scan : record { Variables needed during a scan. }
      accum : real; { Accumulator for A/D data (COR & POW). }
      cycle : integer; { Cycles so far (use with par.skip). }
      mode : integer; { Flag indicating status of scan: 0 = not scanning;
        1 or more: internal codes depending on dt. See ScanCheck
        for details. }
{       num : integer; { Number of scans so far. }
{       pt : integer; { Point number (for dt_COR & dt_POW scans). }
      shots : integer; { Number of shots so far. }
      starttime : longint; { Time (in 1/100 sec) of most recent scan number
        (dt_ELE, dt_MASS) or point (dt_COR, dt_POW). }
      steptime : longint; { Time to wait between scans or shots, depending
        on dt (see starttime). }
    end;
    sel : integer; { Flag indicating selection. }
    tmp : array[1 .. MAXPOINTS] of integer; { Data from last read -- for
      background subtraction. }
    time1 : real; { Storage of 1st time point during mass calibration. }
    vmin : real; { V value (converted intensity) of smallest point in wave;
      this depends on xaxismode (recalculated whenever xaxismode changed). }
    vmax : real; { V value of largest point in wave. See ymin. }
  end;
```

```
var
  pardf : array[1 .. dt_MAX] of par_type; { Default par
    values. }
  wv : record { Waves. }
    cur : integer; { Current wave. }
    ls : array[1 .. MAXWAVES] of ^wave_type; { Wave list. }
    num : integer; { Number of waves loaded. }
    sel : integer; { Flag if any waves selected by get_wv. }
    temp : array[1 .. MAXPOINTS] of real; { Extra wave for general
      use (sm, etc.). }
  end;

implementation

end.
```

4.8. dosshell.pas

```
unit DOSShell;

{ Set of routines designed to simulate DOS commands from within a Pascal
  program. Should be compatible with any program. }

interface

uses
  crt, dos, KEYS;

procedure ShellToDOS; { Main routine. }

implementation

const
  BLANK = '
';
    { 80 chars. }
  ERROR = 'DOS Error.' + BELL;

procedure cd(s : string);
{ Change directory to s.}
begin
  while s[1] = ' ' do { Skip spaces. }
    s := copy(s, 2, length(s));
  {$I-} { Turn off error checking. }
  ChDir(s);
  {$I+} { Turn back on. }
  if IoResult <> 0 then
    writeln(ERROR);
end;

procedure dir_entry(s : SearchRec); forward;

procedure dir;
{ Print current directory contents. }
const
  FILES = '*.*';
  ATTRIB = ANYFILE;
  NUMLINES = 23;
var
  i : integer;
  s : SearchRec;
begin
  FindFirst(FILES, ATTRIB, s);
  while true do
  begin
    for i := 1 to NUMLINES do
    begin
      if DosError > 0 then
        exit;
      dir_entry(s);
      findnext(s);
    end;
    write('Press any key to continue.');
    case readkey of
      CTRLC : { Get out. }
        begin
          writeln;
          exit;
        end;
      EXTENDED : readkey;
    end;
    { Blank out message. }
    gotoxy(1, wherey);
    write(copy(BLANK, 1, 79));
    gotoxy(1, wherey);
  end;
end;

procedure dir_entry(s : SearchRec);
{ Print directory entry. }
var
  dt : DateTime;
  s2 : string;
begin
  write(s.name, copy(BLANK, 1, 16 - length(s.name)));
  if s.attr and DIRECTORY > 0 then
    writeln('<DIR>')
  else
  begin
    str(s.size, s2);
    write(s2, copy(BLANK, 1, 12 - length(s2)));
    UnpackTime(s.time, dt);
    with dt do
    begin
      str(month + 100, s2);
      write(copy(s2, 2, 2), '-');
      str(day + 100, s2);
      write(copy(s2, 2, 2), '-');
      write(year, '    ');
      str(hour + 100, s2);
      write(copy(s2, 2, 2), ':');
      str(min + 100, s2);
      write(copy(s2, 2, 2), ':');
      str(sec + 100, s2);
      writeln(copy(s2, 2, 2));
    end;
  end;
end;

{***************************MAIN ROUTINE*****************************}
procedure ShellToDOS;
var
  s : string;
```

```
begin
  writeln('Stripped-down DOS shell. Not all DOS commands supported!');
  writeln('Type EXIT to return to program.');
  while true do
  begin
    getdir(0, s);
    write('SHELL ' + s + '>');
    readln(s);
    if (length(s) > 1) and (s[2] = ':') then { Change drive. }
      cd(s)
    else if copy(s, 1, 2) = 'cd' then { Change directory. }
      cd(copy(s, 3, length(s)))
    else if s = 'dir' then { Print contents of current directory. }
      dir
    else if s = 'exit' then
      exit
    else if s <> '' then
      writeln(ERROR);
  end;
end;

end.
```

4.9. keys.pas

```
unit keys;

interface

const
{ Normal keys: }
  ACK = #23; { Service request acknowledge. }
  BELL = #7; { Bell. }
  BS = #8; { Backspace. }
  ESC = #27; { Escape. }
  EXTENDED = #0; { Extended key (see below). }
  LF = #10; { Linefeed. }
  CR = #13; { Carriage return. }
  TAB = #9; { Tab. }
  CTRLA = #1;
  CTRLB = #2;
  CTRLC = #3;
  CTRLD = #4;
  CTRLE = #5;
  CTRLF = #6;
  CTRLG = #7;
  CTRLH = #8;
  CTRLI = #9;
  CTRLJ = #10;
  CTRLK = #11;
  CTRLL = #12;
  CTRLM = #13;
  CTRLN = #14;
  CTRLO = #15;
  CTRLP = #16;
  CTRLQ = #17;
  CTRLR = #18;
  CTRLS = #19;
  CTRLT = #20;
  CTRLU = #21;
  CTRLV = #22;
  CTRLW = #23;
  CTRLX = #24;
  CTRLY = #25;
  CTRLZ = #26;

{ Extended keys: }
  XALTA = #30;
  XALTB = #48;
  XALTC = #46;
  XALTD = #32;
  XALTE = #18;
  XALTF = #33;
  XALTG = #34;
  XALTH = #35;
  XALTI = #23;
  XALTJ = #36;
  XALTK = #37;
  XALTL = #38;
  XALTM = #50;
  XALTN = #49;
  XALTO = #24;
  XALTP = #25;
  XALTQ = #16;
  XALTR = #19;
  XALTS = #31;
  XALTT = #20;
  XALTU = #22;
  XALTV = #47;
  XALTW = #17;
  XALTX = #45;
  XALTY = #21;
  XALTZ = #44;
  XALT1 = #120;
  XALT2 = #121;
  XALT3 = #122;
  XALT4 = #123;
  XALT5 = #124;
  XALT6 = #125;
  XALT7 = #126;
  XALT8 = #127;
  XALT9 = #128;
  XALT0 = #129;
  XALTMINUS = #130;
  XALTPLUS = #131;
  XARROWDOWN = #80;
  XARROWLEFT = #75;
  XARROWRIGHT = #77;
  XARROWUP = #72;
  XCTRLARROWLEFT = #115;
  XCTRLARROWRIGHT = #116;
  XCTRLEND = #117;
  XCTRLHOME = #119;
  XCTRLPAGEDOWN = #118;
  XCTRLPAGEUP = #132;
  XDELETE = #83;
  XEND = #79;
  XF1 = #59;
  XF2 = #60;
  XF3 = #61;
  XF4 = #62;
  XF5 = #63;
```

```
  XF6 = #64;
  XF7 = #65;
  XF8 = #66;
  XF9 = #67;
  XF10 = #68;
  XF11 = #133;
  XF12 = #134;
  XHOME = #71;
  XINSERT = #82;
  XPAGEDOWN = #81;
  XPAGEUP = #73;
  XSHIFTF1 = #84;
  XSHIFTF2 = #85;
  XSHIFTF3 = #86;
  XSHIFTF4 = #87;
  XSHIFTF5 = #88;
  XSHIFTF6 = #89;
  XSHIFTF7 = #90;
  XSHIFTF8 = #91;
  XSHIFTF9 = #92;
  XSHIFTF10 = #93;

implementation

end.
```

4.10. tpdecl.pas

```
(************************* Turbo Pascal Declarations *********************)
unit tpdecl;

{$l tpib}

interface
Const

(* GPIB Commands:                                                        *)

   UNL  =  $3f;                    (* GPIB unlisten command             *)
   UNT  =  $5f;                    (* GPIB untalk command               *)
   GTL  =  $01;                    (* GPIB go to local                  *)
   SDC  =  $04;                    (* GPIB selected device clear        *)
   PPC  =  $05;                    (* GPIB parallel poll configure      *)
   GET  =  $08;                    (* GPIB group execute trigger        *)
   TCT  =  $09;                    (* GPIB take control                 *)
   LLO  =  $11;                    (* GPIB local lock out               *)
   DCL  =  $14;                    (* GPIB device clear                 *)
   PPU  =  $15;                    (* GPIB parallel poll unconfigure    *)
   SPE  =  $18;                    (* GPIB serial poll enable           *)
   SPD  =  $19;                 (* GPIB serial poll disable          *)
   PPE  =  $60;                   (* GPIB parallel poll enable        *)
   PPD  =  $70;                   (* GPIB parallel poll disable       *)

(* GPIB status bit vector:                                               *)

   ERR   = $8000;                    (* Error detected                 *)
   TIMO  = $4000;                    (* Timeout                        *)
   EEND  = $2000;                    (* EOI or EOS detected            *)
   SRQI  = $1000;                    (* SRQ detected by CIC            *)
   RQS   = $800;                     (* Device needs service           *)
   SPOLL = $400;                     (* Board has been serially polled *)
   EVENT = $200;                     (* An event has occured           *)
   CMPL  = $100;                     (* I/O completed                  *)
   LOK   = $80;                      (* Local lockout state            *)
   REM   = $40;                      (* Remote state                   *)
   CIC   = $20;                      (* Controller-in-Charge           *)
   ATN   = $10;                      (* Attention asserted             *)
   TACS  = $8;                       (* Talker active                  *)
   LACS  = $4;                       (* Listener active                *)
   DTAS  = $2;                       (* Device trigger state           *)
   DCAS  = $1;                       (* Device clear state             *)

(* Error messages returned in global variable IBERR:                     *)

   EDVR  = 0;                   (* DOS error                           *)
   ECIC  = 1;                        (* Function requires GPIB board to be CIC
*)
   ENOL  = 2;                        (* Write function detected no Listeners
*)
   EADR  = 3;                     (* Interface board not addressed correctly  *)
   EARG  = 4;                     (* Invalid argument to function call        *)
   ESAC  = 5;                        (* Function requires GPIB board to be SAC
*)
   EABO  = 6;                        (* I/O operation aborted
*)
   ENEB  = 7;                        (* Non-existent interface board
*)
   EOIP  = 10;                       (* I/O operation started before previous
*)
                                     (* operation completed
*)
   ECAP  = 11;                       (* No capability for intended operation
*)
   EFSO  = 12;                       (* File system operation error
*)
   EBUS  = 14;                       (* Command error during device call
*)
   ESTB  = 15;                       (* Serial poll status byte lost
*)
   ESRQ  = 16;              (* SRQ remains asserted                    *)
   ETAB  = 20;                 (* The return buffer is full                *)
   ELCK  = 21;

(* EOS mode bits:                                                        *)

   BIN  = $1000;                     (* Eight bit compare              *)
   XEOS = $800;                      (* Send EOI with EOS byte         *)
   REOS = $400;                      (* Terminate read on EOS          *)

(* Timeout values and meanings:                                          *)

   TNONE    = 0;                     (* Infinite timeout  (disabled)   *)
   T10us    = 1;                     (* Timeout of 10 us  (ideal)      *)
   T30us    = 2;                     (* Timeout of 30 us  (ideal)      *)
   T100us   = 3;                     (* Timeout of 100 us (ideal)      *)
   T300us   = 4;                     (* Timeout of 300 us (ideal)      *)
   T1ms     = 5;                     (* Timeout of 1 ms   (ideal)      *)
   T3ms     = 6;                     (* Timeout of 3 ms   (ideal)      *)
   T10ms    = 7;                     (* Timeout of 10 ms  (ideal)      *)
   T30ms    = 8;                     (* Timeout of 30 ms  (ideal)      *)
   T100ms   = 9;                     (* Timeout of 100 ms (ideal)      *)
   T300ms   = 10;                    (* Timeout of 300 ms (ideal)      *)
   T1s      = 11;                    (* Timeout of 1 s    (ideal)      *)
```

```
   T3s      = 12;                    (* Timeout of 3 s    (ideal)      *)
   T10s     = 13;                    (* Timeout of 10 s   (ideal)      *)
   T30s     = 14;                    (* Timeout of 30 s   (ideal)      *)
   T100s    = 15;                    (* Timeout of 100 s  (ideal)      *)
   T300s    = 16;                    (* Timeout of 300 s  (ideal)      *)
   T1000s   = 17;                    (* Timeout of 1000 s (maximum)    *)

(*  IBEVENT Constants                                                   *)

   EventDTAS =  1;
   EventDCAS =  2;

(* Miscellaneous:                                                      *)

   S  = $08;                      (* Parallel poll sense bit          *)
   LF = $0A;                             (* ASCII linefeed character      *)

   NO_SAD  = 0;                     (* test only the pad               *)
   ALL_SAD = -1;                    (* test all secondary addresses of pad *)

(* Values for the 488.2 Send command                                   *)

   NULLend = 00;                 (* do nothing at the end of a transfer  *)
   NLend   = 01;                 (* send NL with EOI after a transfer    *)
   DABend  = 02;                 (* send EOI with the last DAB           *)

(* This value is used by the 488.2 Receive command.                    *)

   STOPend = $0100;              (* stop the read on EOI                 *)


(* This value is used to terminate a 488.2 address list.  It should be  *)
(* assigned to the pad field of the last entry.                         *)

   NOADDR = -1;

(* The following constants are used for the second parameter of the     *)
(* ibconfig function.  They are the "option" selection codes.           *)

   IbcPAD=          $0001;  (* Primary Address                         *)
   IbcSAD=          $0002;  (* Secondary Address                       *)
   IbcTMO=          $0003;  (* Timeout Value                           *)
   IbcEOT=          $0004;  (* Send EOI with last data byte?           *)
   IbcPPC=          $0005;  (* Parallel Poll Configure                 *)
   IbcREADDR=       $0006;  (* Repeat Addressing                       *)
   IbcAUTOPOLL=     $0007;  (* Disable Auto Serial Polling             *)
   IbcCICPROT=      $0008;  (* Use the CIC Protocol?                   *)
   IbcIRQ=          $0009;  (* Use PIO for I/O                         *)
   IbcSC=           $000A;  (* Board is System Controller.             *)
   IbcSRE=          $000B;  (* Assert SRE on device calls?             *)
   IbcEOSrd=        $000C;  (* Terminate reads on EOS.                 *)
   IbcEOSwrt=       $000D;  (* Send EOI with EOS character.            *)
   IbcEOScmp=       $000E;  (* Use 7 or 8-bit EOS compare.             *)
   IbcEOSchar=      $000F;  (* The EOS character.                      *)
   IbcPP2=          $0010;  (* Use Parallel Poll Mode 2.               *)
   IbcTIMING=       $0011;  (* NORMAL, HIGH, or VERY_HIGH timing.      *)
   IbcDMA=          $0012;  (* Use DMA for I/O.                        *)
   IbcReadAdjust=   $0013;  (* Swap bytes during an ibrd.              *)
   IbcWriteAdjust=  $0014;  (* Swap bytes during an ibwrt.             *)
   IbcEventQueue=   $0015;  (* Enable/disable the event queue.         *)
   IbcSPollBit=     $0016;  (* Enable/disable the visibility of SPOLL. *)
   IbcSendLLO=      $0017;  (* Enable/disable the sending of LLO.      *)
   IbcSPollTime=    $0018;  (* Set the timeout value for serial polls. *)
   IbcPPollTime=    $0019;  (* Set the parallel poll length period.    *)
   IbcNoEndBitOnEOS= $001A; (* Remove EOS from END bit of IBSTA.       *)
   IbcUnAddr=       $001B;  (* Enable/disable device unaddressing.     *)

(* Constants that can be used (in addition to the ibconfig constants)  *)
(* when calling the IBASK function.                                    *)

   IbaPAD          = $0001;     (* Primary Address                     *)
   IbaSAD          = $0002;     (* Secondary Address                   *)
   IbaTMO          = $0003;     (* Timeout Value                       *)
   IbaEOT          = $0004;     (* Send EOI with last data byte?       *)
   IbaPPC          = $0005;     (* Parallel Poll Configure             *)
   IbaREADDR       = $0006;     (* Repeat Addressing                   *)
   IbaAUTOPOLL     = $0007;     (* Disable Auto Serial Polling         *)
   IbaCICPROT      = $0008;     (* Use the CIC Protocol?               *)
   IbaIRQ          = $0009;     (* Use PIO for I/O                     *)
   IbaSC           = $000A;     (* Board is System Controller.         *)
   IbaSRE          = $000B;     (* Assert SRE on device calls?         *)
   IbaEOSrd        = $000C;     (* Terminate reads on EOS.             *)
   IbaEOSwrt       = $000D;     (* Send EOI with EOS character.        *)
   IbaEOScmp       = $000E;     (* Use 7 or 8-bit EOS compare.         *)
   IbaEOSchar      = $000F;     (* The EOS character.                  *)
   IbaPP2          = $0010;     (* Use Parallel Poll Mode 2.           *)
   IbaTIMING       = $0011;     (* NORMAL, HIGH, or VERY_HIGH timing.  *)
   IbaDMA          = $0012;     (* Use DMA for I/O.                    *)
   IbaReadAdjust   = $0013;     (* Swap bytes during an ibrd.          *)
   IbaWriteAdjust  = $0014;     (* Swap bytes during an ibwrt.         *)
   IbaEventQueue   = $0015;     (* Enable/disable the event queue.     *)
   IbaSPollBit     = $0016;     (* Enable/disable the visibility of SPOLL. *)
   IbaSendLLO      = $0017;     (* Enable/disable the sending of LLO.  *)
   IbaSPollTime    = $0018;     (* Set the timeout value for serial polls. *)
   IbaPPollTime    = $0019;     (* Set the parallel poll length period *)
   IbaNoEndBitOnEos = $001A;    (* Remove EOS from END bit of IBSTA.   *)
   IbaUnAddr       = $001B;     (* Enable/disable device unaddressing. *)
   IbaBNA          = $0200;     (* A device's access board.            *)
   IbaBaseAddr     = $0201;     (* A GPIB board's base I/O address.    *)
   IbaDmaChannel   = $0202;     (* A GPIB board's DMA channel.         *)
   IbaIrqLevel     = $0203;     (* A GPIB board's IRQ level.           *)
   IbaBaud         = $0204;     (* Baud rate used to communicate to CT box. *)
   IbaParity       = $0205;     (* Parity setting for CT box.          *)
   IbaStopBits     = $0206;     (* Stop bits used for communicating to CT. *)
   IbaDataBits     = $0207;     (* Data bits used for communicating to CT. *)
   IbaComPort      = $0208;     (* System COM port used for CT box.    *)
   IbaComIrqLevel  = $0209;     (* System COM port's interrupt level.  *)
   IbaComPortBase  = $020A;     (* System COM port's base I/O address. *)

(*  The following values are used by the iblines function.  The integer *)
(*    returned by iblines contains:                                     *)
(*         The lower byte will contain a "monitor" bit mask.  If a bit  *)
(*              is set (1) in this mask, then the corresponding line    *)
(*              can be monitored by the driver.  If the bit is clear (0), *)
(*              then the line cannot be monitored.                      *)
(*       The upper byte will contain the status of the bus lines.       *)
(*              Each bit corresponds to a certain bus line, and has     *)
(*              a corresponding "monitor" bit in the lower byte.        *)

   ValidEOI  = $0080;
   ValidATN  = $0040;
   ValidSRQ  = $0020;
   ValidREN  = $0010;
   ValidIFC  = $0008;
   ValidNRFD = $0004;
   ValidNDAC = $0002;
```

```
   ValidDAV  = $0001;
   BusEOI    = $8000;
   BusATN    = $4000;
   BusSRQ    = $2000;
   BusREN    = $1000;
   BusIFC    = $0800;
   BusNRFD   = $0400;
   BusNDAC   = $0200;
   BusDAV    = $0100;

(*************************************************************************)

    nbufsize = 7;    (* Length of board/device names -- hard-coded in TPIB *)
    flbufsize = 50;  (* A generous length for filenames -- the
                        minimum allowed by the handler is 32.
                          50 is hard-coded in TPIB *)
    bufsize = 255;   (* Length of read/write buffer *)
(*************************************************************************)

Type cbuf  = packed array[1..255] of char;   (* character buffer          *)
     nbuf  = packed array[1..7]  of char;    (* board/device name         *)
     flbuf = packed array[1..50] of char;    (* file name                 *)
     AddrList = array[0..255] of integer;    (* 488.2 address list        *)
     srqproc = ^longInt;                     (* srq procedure             *)

(*************************************************************************)

var ibsta : integer;                    (* status word
*)
var iberr : integer;                    (* GPIB error code
*)
var ibcnt : integer;                    (* number of bytes sent or DOS error
*)
var ibcntl: longint;                    (* number of bytes sent or DOS error
*)

(*************************************************************************)
(* The following variables may be used directly in your application program.*)
Var
    bname  : nbuf;                              (* board name buffer
*)
    bdname : nbuf;                              (* board or device name buffer
*)
    flname : flbuf;                         (* filename buffer              *)
    wrt    : cbuf;                          (* write buffer                 *)
    rd     : cbuf;                          (* read buffer                  *)

(* The GPIB board functions declared public by TPIB.OBJ:                    *)

   procedure  ibask  (bd:integer; option:integer; var retval:integer);
   procedure  ibbna  (bd:integer; var bname);
   procedure  ibcac  (bd:integer; v:integer);
   procedure  ibclr  (bd:integer);
   procedure  ibcmd  (bd:integer; var cmd; cnt:longInt);
   procedure  ibcmda (bd:integer; var cmd; cnt:longInt);
   procedure  ibconfig (bd:integer; option:integer; v:integer);
   procedure  ibdiag (bd:integer; var rd; cnt:longInt);
   function   ibdev  (boardID:integer; pad:integer; sad:integer; tmo:integer;
                      eot:integer; eos:integer):integer;
   procedure  ibdma  (bd:integer; v:integer);
   procedure  ibeos  (bd:integer; v:integer);
   procedure  ibeot  (bd:integer; v:integer);
   procedure  ibevent (bd:integer; var event:integer);
```

```
   function   ibfind  (var bdname):integer;
   procedure  ibgts   (bd:integer; v:integer);
   procedure  ibist   (bd:integer; v:integer);
   procedure  iblines (bd:integer; var lines:integer);
   procedure  ibln    (bd:integer; pad:integer; sad:integer;
                       var listen:integer);
   procedure  ibloc   (bd:integer);
   procedure  ibnil;
   procedure  ibonl   (bd:integer; v:integer);
   procedure  ibpad   (bd:integer; v:integer);
   procedure  ibpct   (bd:integer);
   procedure  ibpoke  (bd:integer; option:integer; v:integer);
   procedure  ibppc   (bd:integer; v:integer);
   procedure  ibrd    (bd:integer; var rd; cnt:longInt);
   procedure  ibrda   (bd:integer; var rd; cnt:longInt);
   procedure  ibrdf   (bd:integer; var flname);
   procedure  ibrpp   (bd:integer; var ppr:integer);
   procedure  ibrsc   (bd:integer; v:integer);
   procedure  ibrsp   (bd:integer; var spr:integer);
   procedure  ibrsv   (bd:integer; v:integer);
   procedure  ibsad   (bd:integer; v:integer);
   procedure  ibsic   (bd:integer);
   procedure  ibsre   (bd:integer; v:integer);
   procedure  ibsrq   (srqfunc:srqproc);
   procedure  ibstop  (bd:integer);
   procedure  ibtmo   (bd:integer; v:integer);
   procedure  ibtrap  (mask:word; v:integer);
   procedure  ibtrg   (bd:integer);
   procedure  ibwait  (bd:integer; mask:word);
   procedure  ibwrt   (bd:integer; var wrt; cnt:longInt);
   procedure  ibwrta  (bd:integer; var wrt; cnt:longInt);
   procedure  ibwrtf  (bd:integer; var flname);
   procedure  ibxtrc  (bd:integer; var rd; cnt:integer);
   procedure  ibwrtkey (bd:integer; var wrt; cnt:longInt);
   procedure  ibrdkey (bd:integer; var rd; cnt:longInt);

(********************** The 488.2 entry points ***********************)

   procedure  SendCmds (boardID:integer; var buf; cnt:longInt);
   procedure  SendSetup (boardID:integer; var listen:AddrList);
   procedure  SendDataBytes (boardID:integer; var buf; cnt:longInt;
                eot_mode:integer);
   procedure  Send (boardID:integer; listener:integer;
                var databuf; datacnt:longInt;
                eot_mode:integer);
   procedure  SendList (boardID:integer; var listeners:AddrList;
                var databuf; datacnt:longInt;
                eot_mode:integer);
   procedure  ReceiveSetup (boardID:integer; talker:integer);
   procedure  RcvRespMsg (boardID:integer; var buf; cnt:longInt;
                eot_mode:integer);
   procedure  Receive (boardID:integer; talker:integer; var buf;
                cnt:longInt; eot_mode:integer);
   procedure  SendIFC (boardID:integer);
   procedure  DevClear (boardID:integer; laddr:integer);
   procedure  DevClearList (boardID:integer; var laddrs:AddrList);
   procedure  EnableLocal (boardID:integer; var laddrs:AddrList);
   procedure  EnableRemote (boardID:integer; var laddrs:AddrList);
   procedure  SetRWLS (boardID:integer; var laddrs:AddrList);
   procedure  SendLLO (boardID:integer);
   procedure  PassControl (boardID:integer; talker:integer);
   procedure  ReadStatusByte (boardID:integer; talker:integer;
                var result:integer);
```

```
   procedure  Trigger (boardID:integer; laddr:integer);
   procedure  TriggerList (boardID:integer; var laddrs:AddrList);
   procedure  PPollConfig (boardID:integer; laddr:integer; dataline:integer;
                linesense:integer);
   procedure  PPollUnconfig (boardID:integer; var laddrs:AddrList);
   procedure  PPoll (boardID:integer; var res_ptr:integer);
   procedure  ResetSys (boardID:integer; var laddrs:AddrList);
   procedure  FindRQS (boardID:integer; var taddrs:AddrList;
                var dev_stat:integer);
   procedure  AllSpoll (boardID:integer; var taddrs:AddrList;
                var res:AddrList);
   procedure  FindLstn (boardID:integer; var pads:AddrList;
                var result:AddrList; limit:integer);
   procedure  TestSys (boardID:integer; var addrs:AddrList;
                var result:AddrList);
   procedure  TestSRQ (boardID:integer; var result:integer);
   procedure  WaitSRQ (boardID:integer; var result:integer);


IMPLEMENTATION

var
        our_lcv: integer;      (* local loop control variable       *)


(* The GPIB board functions declared public by TPIB.OBJ:                  *)

   procedure  ibask  (bd:integer; option:integer; var retval:integer); external;
   procedure  ibbna  (bd:integer; var bname); external;
   procedure  ibcac  (bd:integer; v:integer); external;
   procedure  ibclr  (bd:integer); external;
   procedure  ibcmd  (bd:integer; var cmd; cnt:longInt); external;
   procedure  ibcmda (bd:integer; var cmd; cnt:longInt); external;
   procedure  ibconfig (bd:integer; option:integer; v:integer); external;
   procedure  ibdiag (bd:integer; var rd; cnt:longInt); external;
   function   ibdev  (boardID:integer; pad:integer; sad:integer; tmo:integer;
                      eot:integer; eos:integer):integer; external;
   procedure  ibdma  (bd:integer; v:integer); external;
   procedure  ibeos  (bd:integer; v:integer); external;
   procedure  ibeot  (bd:integer; v:integer); external;
   procedure  ibevent (bd:integer; var event:integer); external;
   function   ibfind (var bdname):integer; external;
   procedure  ibgts  (bd:integer; v:integer); external;
   procedure  ibist  (bd:integer; v:integer); external;
   procedure  iblines  (bd:integer; var lines:integer); external;
   procedure  ibln   (bd:integer; pad:integer; sad:integer;
                      var listen:integer); external;
   procedure  ibloc  (bd:integer); external;
   procedure  ibnil; external;
   procedure  ibonl  (bd:integer; v:integer); external;
   procedure  ibpad  (bd:integer; v:integer); external;
   procedure  ibpct  (bd:integer); external;
   procedure  ibpoke (bd:integer; option:integer; v:integer); external;
   procedure  ibppc  (bd:integer; v:integer); external;
   procedure  ibrd   (bd:integer; var rd; cnt:longInt); external;
   procedure  ibrda  (bd:integer; var rd; cnt:longInt); external;
   procedure  ibrdf  (bd:integer; var flname); external;
   procedure  ibrpp  (bd:integer; var ppr:integer); external;
   procedure  ibrsc  (bd:integer; v:integer); external;
   procedure  ibrsp  (bd:integer; var spr:integer); external;
   procedure  ibrsv  (bd:integer; v:integer); external;
   procedure  ibsad  (bd:integer; v:integer); external;
   procedure  ibsic  (bd:integer); external;
   procedure  ibsre  (bd:integer; v:integer); external;
   procedure  ibsrq  (srqfunc:srqproc); external;
   procedure  ibstop (bd:integer); external;
   procedure  ibtmo  (bd:integer; v:integer); external;
   procedure  ibtrap (mask:word; v:integer); external;
   procedure  ibtrg  (bd:integer); external;
   procedure  ibwait (bd:integer; mask:word); external;
   procedure  ibwrt  (bd:integer; var wrt; cnt:longInt); external;
   procedure  ibwrta (bd:integer; var wrt; cnt:longInt); external;
   procedure  ibwrtf (bd:integer; var flname); external;
   procedure  ibxtrc (bd:integer; var rd; cnt:integer); external;
   procedure  ibwrtkey (bd:integer; var wrt; cnt:longInt); external;
   procedure  ibrdkey (bd:integer; var rd; cnt:longInt); external;

(********************** The 488.2 entry points ***********************)

   procedure  SendCmds (boardID:integer; var buf; cnt:longInt); external;
   procedure  SendSetup (boardID:integer; var listen:AddrList); external;
   procedure  SendDataBytes (boardID:integer; var buf; cnt:longInt;
                eot_mode:integer); external;
   procedure  Send (boardID:integer; listener:integer;
                var databuf; datacnt:longInt;
                eot_mode:integer); external;
   procedure  SendList (boardID:integer; var listeners:AddrList;
                var databuf; datacnt:longInt;
                eot_mode:integer); external;
   procedure  ReceiveSetup (boardID:integer; talker:integer); external;
   procedure  RcvRespMsg (boardID:integer; var buf; cnt:longInt;
                eot_mode:integer); external;
   procedure  Receive (boardID:integer; talker:integer; var buf;
                cnt:longInt; eot_mode:integer); external;
   procedure  SendIFC (boardID:integer); external;
   procedure  DevClear (boardID:integer; laddr:integer); external;
   procedure  DevClearList (boardID:integer; var laddrs:AddrList); external;
   procedure  EnableLocal (boardID:integer; var laddrs:AddrList); external;
   procedure  EnableRemote (boardID:integer; var laddrs:AddrList); external;
   procedure  SetRWLS (boardID:integer; var laddrs:AddrList); external;
   procedure  SendLLO (boardID:integer); external;
   procedure  PassControl (boardID:integer; talker:integer); external;
   procedure  ReadStatusByte (boardID:integer; talker:integer;
                var result:integer); external;
   procedure  Trigger (boardID:integer; laddr:integer); external;
   procedure  TriggerList (boardID:integer; var laddrs:AddrList); external;
   procedure  PPollConfig (boardID:integer; laddr:integer; dataline:integer;
                linesense:integer); external;
   procedure  PPollUnconfig (boardID:integer; var laddrs:AddrList); external;
   procedure  PPoll (boardID:integer; var res_ptr:integer); external;
   procedure  ResetSys (boardID:integer; var laddrs:AddrList); external;
   procedure  FindRQS (boardID:integer; var taddrs:AddrList;
                var dev_stat:integer); external;
   procedure  AllSpoll (boardID:integer; var taddrs:AddrList;
                var res:AddrList); external;
   procedure  FindLstn (boardID:integer; var pads:AddrList;
                var result:AddrList; limit:integer); external;
   procedure  TestSys (boardID:integer; var addrs:AddrList;
                var result:AddrList); external;
   procedure  TestSRQ (boardID:integer; var result:integer); external;
   procedure  WaitSRQ (boardID:integer; var result:integer); external;


begin

        ibsta:=0; (*  initialize global status variables *)
```

```
          iberr:=0;
          ibcnt:=0;
        ibcntl:=0;

          for our_lcv:=1 to nbufsize do (* blank fill name buffers *)
        begin
                  bname[our_lcv]  := ' ';
                bdname[our_lcv] := ' ';
        end;

          for our_lcv:=1 to flbufsize do  (* blank file name buffer *)
                  flname[our_lcv] := ' ';

          for our_lcv:=1 to bufsize do  (* blank read/write buffers *)
        begin
                  wrt[our_lcv] := ' ';
                rd[our_lcv]  := ' ';
        end;
end.
```

4.11. fs.bat

```
mode com2:96,n,8,1,p
fpes
```

4.12. mass.par

```
127 0 1000 I
18 0 2 (H2O)
44 0 100 (CO2)
```

Appendix 2. Femtosecond photoelectron spectrum simulation program (`alpine`, `trans`)

The `alpine` and `trans` programs are designed to simulate a femtosecond photoelectron spectrum at different pump-probe time delays. The `alpine` program (named for the similarity of a wavepacket propagating on a potential energy surface to a skier on an alpine slope) is the main engine, propagating wavepackets and calculating overlap integrals. `trans` (named for Fourier transform, the primary operation performed) simply takes the output from `alpine` and generates photoelectron spectra at different time delays. Because of the way the simulation works, `alpine` need only be run once for each pair of potential surfaces coupled by the probe pulse, and spectra can be obtained at any time delay needed using `trans`. The dimensionality of the potentials is also irrelevant to `trans`, so that a 2- or higher-dimensional simulation (using an improved version of `alpine`) can be used to generate photoelectron spectra without modification to `trans`.

1. Theoretical background

Simulations of photoelectron spectra employ a method first described in 1996.[1] Although the general approach is the same, significant improvements in the efficiency of the calculation have been achieved since that time, which were discussed briefly in Ref. 2. These improvements will be covered in more detail at the end of this section.

Three potential surfaces are needed for the simulation: the anion ground state, anion excited state, and neutral state. In practice, to obtain an accurate photoelectron spectrum, several neutral states must be included, running a separate propagation for each one, and summing the results using appropriate cross sections.[3] The propagation method

is based on Kosloff,[4,5] using a program framework modified from Bradforth.[6] Wavefunctions are represented on an evenly-spaced grid spanning the interesting portions of all three potential energy surfaces, and consisting of a number of points equal to a power of 2 (for compatibility with the fast Fourier transform algorithm). Propagation of wavefunctions is carried out using second-order differencing, with a Fourier transform method of evaluating the kinetic energy term of the Hamiltonian. Details of this scheme may be found in Bradforth.

The pump pulse couples the anion ground and excited states. Generally, the probe pulse couples the anion excited state with the neutral state; however, it is also possible to couple the anion ground state with the neutral state, which is sometimes necessary to consider since the ground state can be perturbed by a strong pump pulse excitation.[7,8] This occurs when a significant population is transferred by the pump pulse to the excited state. The resulting ground state depletion creates a nonstationary wavepacket there, causing deviations from the linear approximation to the calculated excited state wavepacket, as well as creating dynamics on the ground state which have been observed in the FPES spectra, since the probe pulse has sufficient energy to detach electrons from the ground state of I_2^-. These are illustrated, for instance, in Zanni *et al.*[9].

The propagations are carried out in two steps. The pump pulse couples the anion ground ("A") and excited ("B") states only, so a Hamiltonian representing this interaction is used to propagate wavepackets on these two surfaces simultaneously:

$$H = \begin{pmatrix} H_A & -\mu_{AB}E^*_{pu}(t) \\ -\mu_{AB}E_{pu}(t) & H_B \end{pmatrix} \tag{1}$$

$$i\hbar\frac{d}{dt}\begin{pmatrix} |\psi_A(t)\rangle \\ |\psi_B(t)\rangle \end{pmatrix} = H\begin{pmatrix} |\psi_A(t)\rangle \\ |\psi_B(t)\rangle \end{pmatrix}. \tag{2}$$

Here $H_A = T_A + V_A$ and $H_B = T_B + V_B$ are the diagonal Hamiltonians of the ground and excited states; μ_{AB} is the transition dipole moment; $E_{pu}(t) = E_{pu}\,\mathrm{sech}\left(\frac{t}{f\tau_{pu}}\right)e^{i\omega_{pu}t}$ is the electric field of the pump pulse with maximum amplitude E_{pu}, full width at half-maximum (FWHM) τ_{pu}, proportionality factor $f = \frac{1}{2}\mathrm{sech}^{-1}\sqrt{2} \approx 0.5673$, and central frequency ω_{pu}; $|\psi_A\rangle$ and $|\psi_B\rangle$ are the ground and excited state wavefunctions. Note that the ground state wavefunction must be propagated because the pump pulse can easily transfer a significant population to the excited state (as confirmed experimentally in our spectra), thus prohibiting the use of a perturbative treatment, i.e. assuming $|\psi_A\rangle$ does not change. Including this extra step is only slightly more demanding numerically, however, because the majority of the calculational effort is spent propagating on the neutral surface, as described next.

This establishes the formation and evolution of the anion excited state. Generating a photoelectron spectrum involves the interaction of the probe pulse with a nonstationary wavepacket, which is unlike conventional photoelectron spectroscopy where the starting wavepacket is assumed to be in an eigenstate. Therefore, integration over the probe pulse duration must be performed again. Since, in general, the probe pulse is much weaker (~1/10) than the pump, first-order perturbation theory can be used to describe the process. Recognizing that the energy of the probe photon $h\nu_2$ is divided between the energy of the neutral ("C") molecule E_C and the detached photoelectron kinetic energy ε, the total energy of the system E_C^{tot} may be written as:

$$E_C^{tot} = E_B + h\nu_2 = E_C + \varepsilon. \tag{3}$$

where E_B is the energy on surface B. Within this framework, the neutral wavefunction $|\psi_C\rangle$ created by a transition from the anion excited state is given by

$$\left|\psi_C(t;\varepsilon,\Delta t)\right\rangle = \frac{i\mu_{BC}}{\hbar}\int_{-\infty}^{\infty} dt' e^{-i(H_C+\varepsilon)(t-t')/\hbar} E_{\mathrm{pr}}(t'-\Delta t)\left|\psi_B(t')\right\rangle, \tag{4}$$

where Δt is the time delay between pump and probe pulses, as measured from peak intensity times; μ_{BC} is the transition dipole moment between anion excited and neutral states; $H_C = T_C + V_C$ is the neutral diagonal Hamiltonian, and $E_{\mathrm{pr}}(t) = E_{\mathrm{pr}}\,\mathrm{sech}\left(\frac{t}{f\tau_{\mathrm{pr}}}\right)e^{i\omega_{\mathrm{pr}}t}$ is the electric field of the probe pulse (maximum amplitude E_{pr}, FWHM τ_{pr}, central frequency ω_{pr}).

Since, as indicated in the equation, the neutral state is dependent upon the photoelectron energy ε and time delay Δt, determining the intensity of photoelectrons $P(\varepsilon, \Delta t)$ is tantamount to calculating the corresponding neutral wavefunction intensity in the long-time limit:[10,11]

$$P(\varepsilon,\Delta t) = \lim_{t\to\infty}\left\langle\psi_C(t;\varepsilon,\Delta t)\middle|\psi_C(t;\varepsilon,\Delta t)\right\rangle \tag{5}$$

$$= \frac{|\mu_{BC}|^2}{\hbar^2}\int_{-\infty}^{\infty} dt' \int_{-\infty}^{\infty} dt'' e^{i\varepsilon(t'-t'')/\hbar} E_{\mathrm{pr}}{}^*(t''-\Delta t) E_{\mathrm{pr}}(t'-\Delta t)\left\langle\psi_B(t'')\middle|e^{iH_C(t'-t'')/\hbar}\middle|\psi_B(t')\right\rangle \tag{6}$$

$$= \frac{|\mu_{BC}|^2}{\hbar^2}\int_{-\infty}^{\infty} dT e^{i\varepsilon T/\hbar}\left[\int_{t_{\min}}^{t_{\max}} dt'' E_{\mathrm{pr}}{}^*(t''-\Delta t) E_{\mathrm{pr}}(T+t''-\Delta t)\left\langle\chi_C(t'',T)\middle|\psi_B(t''+T)\right\rangle\right] \tag{7}$$

where $T = t' - t''$ represents time spent on surface C; $\left|\chi_C(t'',T)\right\rangle = e^{-iH_CT/\hbar}\left|\psi_B(t'')\right\rangle$ are wavefunctions evolved on surface C which contribute to the overall $|\psi_C\rangle$ wavefunction;

and t_{min} and t_{max} have replaced the infinite integral limits of t'', since in practice the propagation needs only to proceed over a finite, dynamically interesting time range.

The bracketed expression on the last line of the equation is the argument of a Fourier transform, which yields a powerful method for calculating the entire photoelectron spectrum at arbitrary Δt, or even probe pulse shape, without rerunning the propagation. The matrix of overlap integrals between time-dependent wavefunctions on surfaces B and C,

$$O(t'',T) = \left\langle \chi_C(t'',T) \middle| \psi_B(t''+T) \right\rangle \tag{8}$$

is recorded over the course of the anion excited state propagation. To calculate a photoelectron spectrum at a particular Δt, the integral over t'' is evaluated, producing a vector of T-dependent values. This vector is then simply Fourier transformed to produce the photoelectron spectrum $P(\varepsilon, \Delta t)$.

The key advantage of this technique is the great reduction in data storage required for a propagation: rather than storing individual wavefunctions $|\chi_C(t'', T)\rangle$, which take up a prohibitive amount of disk space, especially, for 2- and higher-dimensional grids, only an overlap matrix is needed, which is independent of grid size. Further savings in storage space, as well as computation time, is achieved by recognizing that

$$E_{pr}^*(t'' - \Delta t)E_{pr}(T + t'' - \Delta t) \tag{9}$$

vanishes when T is more than a few times the probe pulse width T_{pr}. Therefore, overlap integrals need only be evaluated for $|T| < nT_{pr}$, $n \approx 3$, effectively making the overlap matrix size scale linearly with propagation time. In addition, by symmetry,

$$O(t'', -T) = O^*(t'' + T, T), \tag{10}$$

requiring overlap integrals to be calculated for positive (or negative) values of T only, achieving another 50% savings in computation time.

Note that the term $e^{i\omega t}$ in both the pump and probe electric field expressions is highly oscillatory, and would normally necessitate a very small propagation time step size in order to faithfully reproduce its time dependence. However, this can be avoided by adding the $e^{i\omega t}$ term to the upper state potential energy (for instance, V_B in the case of the pump pulse), where it functions as an energy shift to the potential. The propagation time step size is therefore limited only by the maximum kinetic energy of the wavefunctions. Note that there are also limitations on the spatial grid spacing as a result of the wavefunction kinetic energy. These points are discussed extensively by Kosloff.[4,5]

Calculations of the wavefunctions $|\chi_C(t'', T)\rangle$ and overlap matrix elements $O(t'', T)$ can be conducted in a number of different ways. The approach taken in `alpine` involves a backward time propagation on surface C, so that previous $|\psi_B\rangle$'s can be integrated with calculated $|\chi_C\rangle$'s while varying T between 0 and a maximum negative value. The procedure for carrying out the calculation is as follows. At a time interval Δt_C:

$$\Delta t_C = \frac{2\pi}{\Delta\varepsilon} f_{\text{safety}}, \tag{11}$$

where $\Delta\varepsilon$ is the range of possible electron kinetic energies, and f_{safety} is an empirical safety factor (≈ 0.4), the current $|\psi_B(t'')\rangle$ is placed at the beginning of a queue of previous $|\psi_B(t'')\rangle$ wavefunctions. The current $|\psi_B(t'')\rangle$ is backward-propagated on the neutral surface for one increment of Δt_C, producing $|\chi_C(t'', T = -\Delta t_C)\rangle$, and an element of the overlap matrix $O(t'', T)$ is evaluated with the queue wavepacket $|\psi_B(t'' - \Delta t_C)\rangle$. This process continues until one column of $O(t'', T)$ has been calculated. The oldest wavefunction $|\psi_B(t'' -$

nT_{pr})> is then thrown out of the queue, and the propagation of $|\psi_B(t'')>$ resumes on the anion excited state surface.

2. Compiling and execution

Compilation of `alpine` and `trans` is straightforward. On a Unix platform such as PC Linux or Silicon Graphics (SGI), the FORTRAN compiler commands `f77` (or `f90`):

```
f77 alpine5.4.1.f -o alpine5.4.1
```

and

```
f77 trans2.2.1.f -o trans2.2.1
```

are generally sufficient to create an executable file (the `-o` option here simply specifies the name of the executable file; without it, the default `a.out` is assigned). Different compilers become finicky over certain areas of the code, for instance, in the handling of strings or numerical precision. As of this writing, both source files have been successfully compiled and tested on the above-named platforms.

Usually `alpine` must be run as a "batch" process, since it will use an hour or more of computer time. The typical format for starting `alpine` as a batch process uses the `nice` command:

```
nice +19 alpine5.4.1 > junk.out &
```

Here the lowest priority (`+19`) has been selected to allow other users full access when logged on. Output has been redirected (>) to the `junk.out` file. This is not really necessary, since the `alpine.out` file is also created, containing the same information, but it prevents the window from continuously scrolling with new updates. The & symbol in-

structs the computer to run the command as a background process, i.e. it will not be terminated when the user logs off.

Because `trans` only takes a couple minutes to execute, it is typically run directly:

```
trans2.2.1
```

Output will appear in the command window, with a copy sent to the `trans.out` file.

3. Input files to `alpine`

3.1. `alpine.inp`

This is the main parameter file for `alpine`. A typical `alpine.inp` file is shown below (line numbers are not part of the file, but are simply used for reference in the text):

Line #

```
1  5410                      Version header
2  1, 1, 1, 0                Save potential surface A, B, C; save with shifts
3  0, 1, 0, 50               Save wavepacket A, B, C, save interval (fs)
4  0, 0                      Eigen decomp flag (0/1) and interval (fs; 0 = auto-
   matic).
5  63.5                      Reduced mass (amu)
6  -200, 400, 0              Propagation time min, max (fs), step (fs; 0 = auto-
   matic)
7  1, 1, 1                   Flags to propagate on surfaces A, B, C
8  2, 0, 0.4                 Surface to use for C prop (1=A,2=B),time step on C
   (0 = automatic), time step safety factor
9  50                        Interval for printing screen statistics (fs)
10 -0.8, 2.7, 50             Chebychev energy min, max (eV), safety factor
11 5                         Potential shelf (eV)
12 0, 0.0                    Surface A vib level, add'l offset (eV)
13 0, 0, 0                   Read wavefunc A from file (0/1), read wavefunc B
   from file (0/1), reverse momentum of B (0/1)
14 2.0, 10.0, 512            Grid min, max (angstroms), number of points
15 1.589, 90.0, 1.0d-4, 0    Pump energy (eV), FWHM (fs), E.mu(AB) - use 0 for no
   coupling, sech/gauss flag (0,1)
16 100, 3                    Probe FWHM (max, fs), multiple of FWHM for limits of
   C overlap matrix
17 0, 4d-6, 60               Abs bdy A flag, factor, num of points
18 0, 0, 10                  Abs bdy B flag, factor, num of points
```

```
19 1                          Pot A: I2-(X2SIGMAu+) (MTZ) (0=r0,D0,beta,V0;
   1=re,we,wexe,V0; 3=file name)
20 3.205,110,0.371,-1.014     r0, we, wexe, V0
21 3                          Pot B (0=r0,D0,beta,V0; 1=re,we,wexe,V0; 3=file
   name)
22 -0.014                     V0
23 3                          Pot C: I2(X) (0=r0,D0,beta,V0; 1=re,we,wexe,V0;
   3=file name)
24 2.855                      V0
```

Note that text appearing after the relevant data on each line is ignored by the program, and so may contain any desired comments. In the example, the names of the variables associated with each input item is shown.

Line 1 contains a version header. This line is compared with the program variable `input_header` to determine if `alpine` can read the file.

Line 2 contains flags indicating whether to save a file with each potential surface A, B and C (`1` indicates yes, and `0`, no; this code is used for all flag variables in the file), and a flag to include the shifts used for eliminating the oscillatory $e^{i\omega t}$ term in the output copy of the potentials.

Line 3 contains a similar set of flags for saving wavepackets $|\psi_A\rangle$, $|\psi_B\rangle$ and $|\chi_C\rangle$, and a time interval (in fs) for saving wavepackets. Note if $|\chi_C\rangle$'s are saved, the program will stop after one propagation on surface C, since this option is used strictly for diagnostic purposes, and saving a set of wavepackets for each C surface propagation would generate a prohibitively large number of files. The wavepacket save interval is automatically rounded to an integer multiple of the base propagation time step (see line 6).

Line 4 contains variables for performing eigenvalue decomposition of $|\psi_A\rangle$. This function is currently not functional, but this line must contain two "dummy" values such as shown in the example.

Line 5 contains the reduced mass of the molecule, in atomic mass units (amu). For a diatomic, this is equal to $m_1m_2/(m_1+m_2)$.

Line 6 contains the time limits and step size of the $|\psi_A\rangle$ and $|\psi_B\rangle$ propagations, in fs. If `0` is given for the step size, it will be calculated automatically from information about the grid size and maximum potential energy of the surfaces. It is seldom necessary to override this automatic option.

Line 7 has flags indicating whether $|\psi_A\rangle$, $|\psi_B\rangle$ and $|\chi_C\rangle$ should be propagated. Usually $|\psi_A\rangle$ is propagated only if explicit inclusion of a ground state depletion effect is needed. By not propagating $|\chi_C\rangle$, the program can be used for simple propagation of anion wavepackets.

Line 8 allows for specification of which anion surface (`1` = A, `2` = B) is to be used to generate $|\chi_C\rangle$, the time step to use on the C surface, and an additional time step "safety factor." As with line 6, the time step can be specified as `0`, meaning it will be automatically calculated from potential parameters. The smaller the time step, the larger the range of electron kinetic energy in the photoelectron spectrum. The safety factor (typically `0.4`) is used to reduce the calculated time step and thereby expand the kinetic energy range slightly, to prevent features near the edges of the spectrum from wrapping around to the other side if they are artificially broad due to a short propagation time.

Line 9 contains the interval (in fs) for printing updates to the screen.

Line 10 contains outdated parameters for the Chebychev propagation scheme, now abandoned. Three "dummy" values must appear, such as shown in the example.

Line 11 contains a potential "shelf" in eV. This is the value each potential surface is truncated to after either reading in or calculating the surface.

Line 12 specifies the initial vibrational level of $|\psi_A\rangle$, and an additional energy offset to surface A (in eV). This offset generally only used when $|\psi_A\rangle$ is read from a file.

Line 13 contains a flag specifying whether to read $|\psi_A\rangle$ and $|\psi_B\rangle$ from files, and a flag to reverse the initial momentum of $|\psi_B\rangle$. If read from files, the filenames must be `psiA.inp` and `psiB.inp`, respectively. The momentum reversal function was added for a special application (taking the final wavefunction of a prior propagation and placing it on a new surface with reversed momentum, to simulate the effect of a wavepacket reflected off a solvent-induced "wall.")

Line 14 specifies the grid parameters: minimum and maximum (in Å), and the number of points. Note this number must be a power of 2.

Line 15 provides information about the the pump pulse: photon energy (in eV), full width at half maximum or FWHM (in fs), the product of $E_{pu} \cdot \mu_{AB}$ (in arbitrary units), and a flag to choose between a sech^2- or Gaussian-shaped pulse (`0` or `1`, respectively).

Line 16 provides information about the probe pulse: FWHM (in fs), and a factor specifying the maximum duration to propagate $|\chi_C\rangle$, in multiples of the probe pulse width (typically `4`). Note that this information is only used to determine propagation parameters for $|\chi_C\rangle$, which defines a maximum possible probe pulse to be used in `trans`.

Lines 17 and 18 contain absorbing boundary condition parameters for surfaces A and B, respectively. For each, there is a flag specifying whether to use the absorbing boundary, a multiplying factor specifying how quickly the boundary drops to zero (use `0` to let `alpine` calculate this automatically), and the number of points over which the absorbing boundary is active.

Lines 19 and 20 contain parameters for the A surface potential. Line 19 contains a flag specifying which type of potential function to use, while line 20 contains the parameters for the potential. The format of line 20 depends on what was provided in line 19:

Value	Potential type	Parameters
0	Morse	r_e (Å), D_0 (eV), β (Å^{-1}), V_0 (eV)
1	Morse	r_e (Å), ω_e (cm^{-1}), $\omega_e x_e$ (cm^{-1}), V_0 (eV)
2	Not used (was LEPS potential)	—
3	Read from file	V_0 (eV)

Here r_e indicates the equilibrium bond distance, D_0 is the well depth, β is the curvature parameter, ω_e is the harmonic frequency, $\omega_e x_e$ is the anharmonicity, and V_0 is the offset energy of the potential (bottom of well assumed 0). Note that the two ways of specifying a Morse potential are equivalent:

$$\beta = \sqrt{\frac{2\mu}{\omega_e x_e}} \text{ and } D_0 = \frac{\omega_e^2}{4\omega_e x_e} \tag{12}$$

where μ is the reduced mass. For potentials read from files, the filenames must be `pot*.in` (* = A, B or C).

Lines 21 and 22 contain the same parameters for surface B.

Lines 23 and 24 contain the same parameters for surface C.

3.2. `pot*.in`

When a potential surface is specified in `alpine.inp` to be read from a file, the filename corresponding to surface A, B or C must be `potA.in`, `potB.in` or `potC.in`, respectively. The format of the files is identical. Line 1 contains grid pa-

rameters of the potential: minimum and maximum (in Å), and the number of points. Note this number must be a power of 2. These parameters need not match the grid specified in `alpine.inp`, as an interpolation feature is built into the program. Subsequent lines contain the potential energy (in eV) for each grid point.

3.3. `psi*.inp`

Previously calculated wavefunctions may be input for $|\psi_A\rangle$ and $|\psi_B\rangle$, with filenames `psiA.inp` and `psiB.inp`, respectively. Each line of the file simply contains one pair of complex numbers indicating the wavefunction amplitude at each grid point, using the standard FORTRAN format:

```
(-1.0000000000, 1.0000000000)
```

where the first number is the real part, and the second number is the imaginary part.

4. Output files from `alpine`

The following files are output during an `alpine` run:

`alpine.out` contains general information about the propagation parameters and periodic updates on the status of the propagation. It is the same information as printed on the screen during the run.

`norm*.out` (* = A, B) contain the norms ($|\langle\psi|\psi\rangle|^2$) of $|\psi_A\rangle$ and $|\psi_B\rangle$, respectively, calculated at time intervals specified by the update status.

`pot*.out` (* = A, B, C) contain the potential functions for each surface, if it was specified to be generated in `alpine.inp`. Each line contains a pair of numbers, distance (in Å) and energy (in eV).

`psi*.dump` (* = A, B) contain $|\psi_A\rangle$ and $|\psi_B\rangle$ at the final time step in complex number format. These are useful for reading back into the propagation code, for instance, to break a long propagation into manageable pieces. This is accomplished by renaming the files `psi*.inp` (see above) and setting the appropriate flags in `alpine.inp`.

`psi*.*`: These are files output for each wavefunction several times during a run, if specified in `alpine.inp`. The first asterisk denotes a letter (`A` or `B`, indicating the wavefunction $|\psi_A\rangle$ or $|\psi_B\rangle$), followed by a three-digit number starting at `001`, and incrementing by 1 each save interval. The second asterisk indicates the file type: `dump` is a complex-number format wavefunction, `real` contains distance (in Å) and the real part of the amplitude, `sq` contains distance (in Å) and squared amplitude, and `mom` contains inverse distance (in $Å^{-1}$) and the real part of the momentum-space amplitude.

`matrix.out` contains the matrix of overlap integrals $O(t'', T)$, in complex-number format, one element to a line. The first line contains the shift of the surface C potential used by `alpine`, which must be applied to the calculation in `trans` to obtain photoelectron spectra with the proper energy offset.

5. Input files to `trans`

5.1. `trans.inp`

The format of the main `trans` input file, `trans.inp`, is very simple, as the following example shows:

Line #

```
1  2200                    Version header
2  4.768, 100              Probe pulse energy (eV) and FWHM (fs)
3  2, 1200, 254            Convolution type, ion beam energy (eV), mass (amu)
```

```
4  1.0, 3.0, 0.005        Output spectrum energy minimum, maximum and step
   size (eV)
5  0                      Time delay (fs) for calculating spectrum
6  100                    etc.
7  200                    etc.
```

As with `alpine.inp` above, text appearing after the relevant data on each line is ignored by the program, and so may contain any desired comments.

Line 1 contains the version header. This line is compared with the program variable `ivers_trans` to determine if `trans` can read the file.

Line 2 contains the probe pulse energy (eV) and FWHM (fs). Note that the FWHM cannot be significantly larger than what was specified in `alpine`, but it may be as small as desired.

Line 3 contains the convolution type, ion beam energy (eV) and ion mass (amu). Currently, five convolution types are supported:

Convolution type	Description	Function (see text)
0	None	–
1	Tophat (flat)	1
2	Isotropic	$\sqrt{1-r}$
3	$\mathrm{Sin}^2\,\theta$	$(1-r)\sqrt{1-r}$
4	$\mathrm{Cos}^2\,\theta$	$r\sqrt{1-r}$

For convolution types > 0, the following parameters are determined from the ion beam energy and mass:

$$U_{\mathrm{off}} = \frac{m_e U_{\mathrm{ion}}}{m_{\mathrm{ion}}} \tag{13}$$

$$r = \frac{\left(U_e - U_e^{\mathrm{cm}} - U_{\mathrm{off}}\right)^2}{4U_{\mathrm{off}}U_e^{\mathrm{cm}}} \tag{14}$$

where the range of the convolution is $\pm 2\sqrt{U_{\text{off}} U_e^{\text{cm}}}$. The energy offset, U_{off}, is called `aeoffset` in the program. The convolution functions (unnormalized) are shown above in the table.

Line 4 contains the output spectrum energy minimum, maximum and step size, all in eV. These parameters may be arbitrary, as the Fourier transform output is always interpolated to fit onto this grid.

Lines 5 and later contain the pump-probe time delays (fs), one to a line. There may be as many time delays as necessary. Calculation of extra time delays is very fast, as the bulk of the computer time is typically spent reading in the `matrix.out` file.

5.2. `matrix.out`

In addition, the `matrix.out` file from `alpine` is read by `trans`.

6. Output files from `trans`

`pw*.out` contain the photoelectron spectrum for each delay time specified by `trans.inp`. The asterisk indicates a three-digit number beginning with `001`, and incrementing by 1 for each subsequent file. The format of each line is the electron kinetic energy (in eV) and the intensity (in arbitrary units).

`trans.out` is simply a copy of what is printed to the screen by `trans`, indicating the progress of the program.

7. References

1 B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, Chem. Phys. Lett. 258, 523 (1996).

2 V. S. Batista, M. T. Zanni, B. J. Greenblatt, D. M. Neumark, and W. H. Miller, J. Chem. Phys. 110, 3736 (1999).

3 M. T. Zanni, V. S. Batista, B. J. Greenblatt, W. H. Miller, and D. M. Neumark, J. Chem. Phys. 110, 3748 (1999).

4 R. Kosloff, J. Phys. Chem. 92, 2087 (1988).

5 R. Kosloff, Annu. Rev. Phys. Chem. 45, 145 (1994).

6 S. E. Bradforth, Ph.D. Thesis, University of California, Berkeley (1992).

7 U. Banin, A. Bartana, S. Ruhman, and R. Kosloff, J. Chem. Phys. 101, 8461-8481 (1994).

8 A. Bartana, U. Banin, S. Ruhman, and R. Kosloff, Chem. Phys. Lett. 229, 211 (1994).

9 M. T. Zanni, T. R. Taylor, B. J. Greenblatt, B. Soep, and D. M. Neumark, J. Chem. Phys. 107, 7613 (1997).

10 C. Meier and V. Engel, J. Chem. Phys. 101, 2673 (1994).

11 C. Meier and V. Engel, Phys. Rev. Lett. 73, 3207 (1994).

8. Program listings

8.1. alpine5.4.1.f

```
C                         A   L   P   I   N   E

C                 1-D femtosecond photoelectron spectrum simulator

C                   Begun by B. Jefferys Greenblatt, August 1995.
C                      Latest version: see last_date variable.

C Adapted from S. E. Bradforth code 'prop10'.

C Address: Neumark Research Group
C          Department of Chemistry
C          University of California
C          Berkeley, CA 94720 USA
C Phone:   510-642-7761 or 510-643-9301
C E-mail:  jeff@radon.cchem.berkeley.edu

C INPUT:  alpine.inp     Input deck (see read_input() for explicit instructions).
C         (pot files)    Names of potential energy files, if used (see potread).
C OUTPUT: alpine.out     Copy of information printed to screen during run.
C         abs.out        Absorbing boundary function (if iabs = 1)
C         normX.out      Norms of wavepackets (X = A, B only) saved at intervals
C                          determined in input deck.
C         potX.out       Surface X potential function (A = anion ground state, B
C                          = anion excited state, C = neutral).
C         psiX.dump      Final wavepackets on surfaces X = A, B only, in raw
C                          (complex) format. Intended for the future use of
C                          reading old wavepackets in, in order to continue a
C                          propagation.
C           psiXY.Z        Wavepackets on surface X (= A, B only) at time interval
C                            Y (= 000, 001, etc.). This is only recorded if the
C                          isavpsiX variable in input deck is set. Information
C                          contained in file indicated by Z: sq = square of psi
C                          vs. position; real = real part of psi vs. position;
C                          dump = real and imag parts of psi (no position); mom
C                          = square of psi vs. momentum.
C         matrix.out     Matrix of overlap integrals: time x time. Read by
C                          Fourier transform program TRANS.

C Notes for this version (add to bottom of list, please):

C 26jul95 BJG: This program calculates photoelectron spectrum intensity over a
C   user-specified range of electron kinetic energies and delay times.

C 18mar96 BJG: Version 2 implemented. This version calculates one matrix con-
C taining all dynamical information for full 2D (delay time x eKE) photoelectron
C spectrum. This matrix, calculated from first order perturbation theory of the
C probe pulse, consists of the final wavefunctions zpsiC1 each of which has been
C promoted to the upper surface at different times t and propagated out to the
C same final time tmax. A Fourier transform of this matrix multiplied by the
C envelope function of the probe pulse, which can vary in delay time relative
C to the probe, followed by calculation of the norms of zpsiC1(w, x) gives the
C photoelectron spectrum P(w) for an arbitrary probe delay (and shape for that
C matter!)

C Currently program uses weak field limit for both pulses since this eliminates
C the need to propagate zpsiA at all (no change to zpsiA). For every time step
C tpsiCstep on surface B, program executes inner loop propagation on surface C
C and saves result in file zpsiCt. Final transformation to useful information
C is handled by another program, trans1.f.

C 28mar96 BJG: Version 2.1 implemented. Replaces second-order differencing
C propagation scheme with Chebychev polynomial approximation for the upper
C state surface only (the first two surfaces are linked by a time-dependent
C coupling term which makes this method difficult to implement; plus, the huge
C majority of the computation time is spent propagating on the upper surface
C anyway). This approach is taken from R. Kosloff, "Propagation Methods for
C Quantum Molecular Dynamics," Annu. Rev. Phys. Chem. 45, 145 (1994). New sub-
C routine chebprop() handles this implementation.

C 7may96 BJG: Version 2.2: Minor change as a test of difference between sech^2
C and Gaussian pulse shapes on dynamics. Is a permanent upgrade, however, as
C allow both types of shapes to be accessed in input deck. Also changed input
C deck filename to alpine.inp, and added a header line with the version number
C of alpine, so we know which version we ran simulation under (important)!

C 8dec96 BJG: Version 3.0 (not a big change from 2.2, but MTZ has made some
C other versions and we don't want to conflict). This solves the small energy
C offset error by changing the sign of the phase factor in the Chebychev propa-
C gator.

C 8dec96 BJG: Version 3.1. Makes shifting around of potentials automatic and
C transparent; shifts effected in ALPINE are undone in TRANS by passing along
C a parameter. Simply set up potentials with true spacings between, v0A can be
C anywhere (program automatically shifts to make v=0 eigenenergy 0).

C 9dec96 BJG: Version 3.2. Timing specified more rationally now, calculates step
C size necessary for FT, added function to read in arbitrary potential with
C automatic interpolation to fit chosen grid.

C 10dec96 BJG: 3.3: Allow recording of momentum representation of wavefunction
C in addition to position, if desired.

C 10dec96 BJG: 3.4: Modified a few lines to allow to compile on Linux machine.

C 15dec96 BJG: 3.4.1: Changed chebprop algorithm in attempt to fix bug when
C using 1024 pts in x grid; however, bug was in the read_user_pot interpolation
C routine (made gap in potential, producing wf spike which blew up propagation).

C 15dec96 BJG: 3.4.2: Kept new chebprop, clean up unused subroutines (including
C dumping off LEPS capability -- so source code file is gone), made screen
C output produce exact copy in alpine.out, including list of letters with cor-
C responding time delays for saved wavefunctions. Added some automatic parame-
C ters such as time steps (both on A/B and C surfaces), option to propagate on B
C or A+B, take FPES from surface A or B.

C 20dec96 BJG: 3.4.3: Allowed initial wavepacket from other than v=0 eigenlevel.
C Combined anionmorsewf and alpine source files, so code is now contained in a
C single file, alpineXXX.f. Changed psiXY.Z labeling scheme to numbers
C (psiA000.sq, etc.).

C 24dec96 BJG: 3.4.4: Added ability to decompose eigenspace of psiA or psiB
C (using Heller scheme: Fourier transform of autocorrelation). [16feb97: Does
C not yet work].

C 16feb97 BJG: 4.0: Switch from recording wavepacket matrix to recording
C overlap matrix, in anticipation of 2-D implementation ("TOBOGGAN").

C 29apr97 BJG: 4.1: Got eigenfunction decomposition working. But then realized
C could simply overlap the final wavefunction with the known Morse
C eigenfunctions of the X state and get the populations much more quickly.
```

```
C Currently it does both -- but with the second approach, there's no need to
C propagate beyond the extend of the pump pulse, so this is the one to use.

C 29apr97 BJG: 4.2: Allow reading and writing of wavepackets to do "special"
C propagations like simulating I2(CO2)16- recurrences at 1.5 ps.

C 30jun97 BJG: 4.3: Allow inhibition of propagation on all surfaces; added A
C surface energy offset independent of nlevel, to inform surface moving algo-
C rithm that initial wavepacket (read from file) has nonzero kinetic energy; re-
C arranged order of some input deck parameters.

C 6jul97 BJG: 5.0: Implemented a dramatic improvement in performance, reducing
C order of propagation time from quadratic to linear in the length of propaga-
C tion! Storage requirements are roughly doubled, however: now must record both
C a C0 and C1 wavefunction for several time slices simultaneously.

C 6jul97 BJG: 5.1: Haven't even tested 5.0 yet, but discovered way to halve
C storage requirements with no decrease in speed, by recording psi2's, rather
C than psi3's, in queue, and propagating C taking overlap every itCstep
C iterations.

C 13aug97 BJG: 5.1.1: Still debugging; this version prints out, starting with
C pw101.out, the un-FFT'd spectra for inspection.

C 1997.8.29 BJG: 5.1.sgi: special SGI source code version which can get
C around the inability to read strings as input.

C 1997.8.29 BJG: 5.2: Again sgi compatible, this just adds a few lines around
C the C prop loops to calculate the number of Hpsi calls needed (j_max).

C 1997.10.1 BJG: 5.3: Absorbing boundary conditions added.

C 1997.10.3 BJG: 5.4: Changed double precision to real*8, added FFT_prep routine
C to improve alpine performance, other small changes for SGI compatibility.

C 1999.02.10 BJG: 5.4.1: Modified so aether can read input headers (numerical
C code, rather than string). Had simply ignored them -- not a great idea.

C Global variable declarations:

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048, nauto = 2048, MAX_QUEUE = 256)
      dimension zpsiA0(npts), zpsiB0(npts), zpsiC0(npts), zpsiC1(npts),
     &  zpsiqueue(npts, MAX_QUEUE), zHpsiA1(npts), zHpsiB1(npts),
     &  zHpsi(npts), zpsiAorig(npts), zovlp(npts)
      common /abss/ absA(npts), absB(npts)
      common /autocorr/ zautoA(nauto)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const1/ xmas, hb, sechfactor, gaussfactor
      common /const2/ x0A, xomegA, v0A, xwexeA, dea, xalphaA, x0B,
     &  xomegB, xwexeB, deB, xalphaB, v0B, x0C, xomegC, xwexeC, deC,
     &  xalphaC, v0C, shelf, v0A_off, v0B_off, v0C_off
      common /const3/ tstep
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      common /const7/ tpktsav, itpktsav, tCstep, itCstep, tCfactor,
     &  emin, emax, factor, e_shift, tmin, tmax, statstep, istatstep,
     &  tauto, itauto
      common /const8/ hv1, fwhm1, dmu1, domega1, sech1, itype1, ipropA,
     &  ipropB, ipropC, iCpot, nlevel, Aoffset, Eint, fwhm2, thresh,
     &  proberange, ireadA, ireadB, irevB
      integer f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /files/ f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /norms/ rnormA, rnormB, rnormC, HavA, HavB, HavC
      common /pot/ zpot(npts, 3)
      common /pottyp/ ipottypA, ipottypB, ipottypC, isavpotA, isavpotB,
     &  isavpotC, isavpsiA, isavpsiB, isavpsiC, ieigenA, ieigenB,
     &  isavpot_shift
      common /zpsis/ zpsiA1(npts), zpsiB1(npts)
      character*80 input_header, last_date, matrix_header
      common /headers/ input_header, last_data, matrix_header
      common /version/ ivers_input, ivers_matrix

C Begin.

      call FFT_prep ! Set up FFT array.
      stab_factor = 0.2 ! Stability factor for propagation.

C Set up header variables -- for compatibility. Used in alpine.inp, matrix.out.
C Must change anytime the file formats change.

      last_date = '1999.02.24'
      ivers_input = 5410
      ivers_matrix = 2200

C File variables -- so only have to change file number in one place.

      f_in = 1
      f_out = 2
      f_normA = 3
      f_normB = 4
      f_matrix = 7
      f_temp = 8
      f_temp2 = 9
      f_temp3 = 10

C Print greeting.

      open(f_out, file = 'alpine.out')

100   format('Welcome to...')
110   format('                          A    L    P    I    N    E')
120   format('           1-D femtosecond photoelectron spectrum ',
     &  'simulator')
130   format('                          by B. Jefferys Greenblatt')
140   format('Last modified ', a)
141   format('Input header = ', i5)
142   format('Matrix header = ', i5)
      write (*, *)
      write (*, 100)
      write (*, *)
      write (*, 110)
      write (*, *)
      write (*, 120)
      write (*, *)
      write (*, 130)
      write (*, *)
      write (*, 140) last_date
      write (*, 141) ivers_input
      write (*, 142) ivers_matrix
      write (f_out, *)
```

```
      write (f_out, 100)
      write (f_out, *)
      write (f_out, 110)
      write (f_out, *)
      write (f_out, 120)
      write (f_out, *)
      write (f_out, 130)
      write (f_out, *)
      write (f_out, 140) last_date
      write (f_out, 141) ivers_input
      write (f_out, 142) ivers_matrix

C Define some useful constants.

      call const()

C Read input deck.

      call read_input()

C Write basic information.

      write (*, *)
      write (f_out, *)

270   format ('xmin =',f9.2,2x, 'xmax =',f9.2,2x, 'nXpts =',i5,2x,
     &  'dx =', e10.4)
      write (*, 270) xmin * a0, xmax * a0, nXpts, dx * a0
      write (f_out, 270) xmin * a0, xmax * a0, nXpts, dx * a0

220   format('xmas/amu = ', f7.2)
      write (*, 220) xmas / amu
      write (f_out, 220) xmas / amu

C Criteria of succesful propagation given in Kosloff, J. Comput. Phys. 52, 35
C (1983); essentially the max kinetic energy representable on a grid with
C spacing dx is given (in au) by pi * pi / (2 * xmas * dx * dx) and the
C stability criterion is tstep * ((pi * pi) / (2 * xmas * dx * dx) + V ) <= 1.0
C (actually best at <= 0.2).

C 16Dec96 BJG: tstep now calculated automatically, unless user specifies (tstep
C > 0 in input deck).

320   format ('Maximum kinetic energy that can be represented is ',
     &  f6.3, ' eV.')
      sqkmax = 4.9348d00 / (xmas * dx * dx)
      write (*, 320) sqkmax * harev
      write (f_out, 320) sqkmax * harev

      if (tstep .eq. 0) then
        tstep = stab_factor / sqkmax
420     format('(tstep automatic)')
        write (*, 420)
        write (f_out, 420)
      endif

300   format ('Propagation time step size = ', f6.3, ' fs.')
      write (*, 300) tstep * atu
      write (f_out, 300) tstep * atu

330   format ('Stability at best, assuming zero potential, is ', f6.3)
      write (*, 330) sqkmax * tstep
      write (f_out, 330) sqkmax * tstep

      if (sqkmax * tstep .gt. 1d0) then
340     format('*****THIS PROPAGATION WILL BE UNSTABLE*****')
        write (*, *)
        write (*, 340)
        write (*, *)
        write (f_out, *)
        write (f_out, 340)
        write (f_out, *)
        stop
      endif

350   format ('Time limits for current propagation = ', f8.2,
     &  ' to ', f8.2, ' fs.')
      write (*, 350) tmin * atu, tmax * atu
      write (f_out, 350) tmin * atu, tmax * atu

C Calculate autocorrelation parameters.

      if (ieigenA .eq. 1) then

        if (tmin .gt. 0) then
531       format('tmin cannot be positive for autocorrelation.')
          write(*, 531)
          write(f_out, 531)
          stop
        endif
        tautomin = -tmin
        trange = tmax - tautomin

C Calculate tauto if = 0, revise interval regardless to equal a multiple of tstep.
C (nint(x + alessthanhalf) rounds up).

        if (tauto .eq. 0) then
530       format('(tauto automatic)')
          write (f_out, 530)
          write (*, 530)
          if (nint(trange / tstep + alessthanhalf) .le. nauto)
     &      then
            tauto = tstep
          else
            tauto = tstep * nint((trange / tstep) / nauto +
     &        alessthanhalf) ! Calculate a suitable multiple of tstep.
          endif
        endif
        itauto = nint(tauto / tstep)
        tauto = tstep * itauto ! Revise value of tauto, if necessary.
        itautomin = nint((tautomin - tmin) / tstep)
        tautomin = tstep * itautomin + tmin ! Revise value of tautomin.

521     format('Revised time interval for recording autocorrelation ',
     &    '(fs) = ', f9.5)
        write (f_out, 521) tauto * atu
        write (*, 521) tauto * atu

522     format('Revised starting time for recording autocorrelation ',
     &    '(fs) = ', f9.5)
        write (f_out, 522) tautomin * atu
        write (*, 522) tautomin * atu

C Check that tauto not so small that we will overrun zautoA during run.

        if (nint(trange / tauto + alessthanhalf) .gt. nauto) then
```

```
511       format('Too many points in autocorrelation. Terminating.')
          write (f_out, 511)
          write (*, 511)
          stop
        endif

C Calculate dephasing constant gamm, for use in autocorrelation.

        gamm = -5.0 / (trange / tauto) ** 2

540     format('Dephasing constant = ', e14.8)
        write(f_out, 540) gamm
        write(*, 540) gamm
      endif

C Calculate interval parameters, revise real intervals, and print revised
C delays.

      itpktsav = nint (tpktsav / tstep)
      tpktsav = itpktsav * tstep
500   format('Revised interval for saving wavepackets = ',
     &  f9.2, ' fs.')
      write (*, 500) tpktsav * atu
      write (f_out, 500) tpktsav * atu

      istatstep = nint(statstep / tstep)
      statstep = istatstep * tstep
510   format('Revised interval for printing statistics = ',
     &  f9.2, ' fs.')
      write (*, 510) statstep * atu
      write (f_out, 510) statstep * atu

C Other miscellaneous parameters.

280   format ('hv/eV =', f7.4, 2x, 'fwhm/fs =', f7.2, 2x, 'E*mu =',
     &   e16.8)
      write (*, 280) hv1 * harev, fwhm1 * atu, dmu1 * a0
      write (f_out, 280) hv1 * harev, fwhm1 * atu, dmu1 * a0

410   format ('Surface generating FPES = ', a1,
     &  '  Propagate surface A = ', i1, ' B = ', i1, ' C = ', i1)
      write (*, 410) char(64 + iCpot), ipropA, ipropB, ipropC
      write (f_out, 410) char(64 + iCpot), ipropA, ipropB, ipropC

      write (*, *)
      write (f_out, *)

C Calculate eigenenergy.

      Eint = xomegA * (nlevel + 0.5) - xwexeA * (nlevel +
     &  0.5) **2
520   format('Initial vibrational level = ', i3, '  Energy = ',
     &  f7.4, ' eV')
      write (*, 520) nlevel, Eint * harev
      write (f_out, 520) nlevel, Eint * harev

C Add energy offset.

      Eint = Eint + Aoffset
550   format('Additional A offset = ', f7.4, ' eV.  Total offset = ',
     &  f7.4, ' eV')
      write (*, 550) Aoffset * harev, Eint * harev
      write (f_out, 550) Aoffset * harev, Eint * harev

C Set up potential offsets based on photon energy, etc:

C   Make starting eigenenergy at 0:

      v0A_off = - Eint - v0A

C   Shift B surface down additionally by photon energy:

      v0B_off = v0A_off - hv1

C   Shift C surface depending on iCpot: iCpot = 1: shift down by average of
C   range of C potential:

      if (iCpot .eq. 1) then
        vdiff_min = v0C + v0A_off
        vdiff_max = zpot(nXpts, 3) + v0A_off
        v0C_off = v0A_off - (vdiff_min + vdiff_max) / 2
      else

C   iCpot = 2: Shift C surface down additionally by average of closest and
C   furthest approaches of potentials:

        vdiff_min = v0C + v0B_off
        vdiff_max = zpot(nXpts, 3) - zpot(nXpts, 2)
        v0C_off = v0B_off - (vdiff_min + vdiff_max) / 2
      endif

C Save the potential surfaces if required.

      if (isavpot_shift .eq. 0) then
150     format ('Saving potentials BEFORE shifting or shelving.')
        write (*, 150)
        write (f_out, 150)
        if (isavpotA .eq. 1) call potlsave (1)
        if (isavpotB .eq. 1) call potlsave (2)
        if (isavpotC .eq. 1) call potlsave (3)
      endif

C Shift potentials and apply shelf condition:

      do ix = 1, nXpts
        zpot(ix, 1) = zpot(ix, 1) + v0A_off
        if (real(zpot(ix, 1)) .gt. shelf) zpot(ix, 1) = shelf
        zpot(ix, 2) = zpot(ix, 2) + v0B_off
        if (real(zpot(ix, 2)) .gt. shelf) zpot(ix, 2) = shelf
        zpot(ix, 3) = zpot(ix, 3) + v0C_off
        if (real(zpot(ix, 3)) .gt. shelf) zpot(ix, 3) = shelf
      enddo

C DEBUG: Save potentials with shifts in effect.

      if (isavpot_shift .eq. 1) then
151     format ('Saving potentials AFTER shifting and shelving.')
        write (*, 151)
        write (f_out, 151)
        if (isavpotA .eq. 1) call potlsave (1)
        if (isavpotB .eq. 1) call potlsave (2)
        if (isavpotC .eq. 1) call potlsave (3)
      endif

C Potential information.
```

```
      write (*, *)
      write (f_out, *)

      if (xwexeA .eq. 0.0d00) then
230     format ('x0A =',f6.2,2x, 'xomegA =',f9.2,2x, 'v0A =',f6.3,
     &    2x, 'v0A_off =', f6.3)
        write (*, 230) x0A * a0, xomegA * harwn, v0A * harev,
     &    v0A_off * harev
        write (f_out, 230) x0A * a0, xomegA * harwn, v0A * harev,
     &    v0A_off * harev
      else
235     format ('x0A =',f6.2,2x, 'xomegA =',f9.2,2x, 'v0A =',f6.3,
     &    2x, 'v0A_off =', f6.3,
     &    /, 2x,'wexeA =',f9.2,2x,'alphaA =',f9.3,2x,'DeA =',f9.3)
42      write (*, 235) x0A * a0, xomegA * harwn, v0A * harev,
     &    v0A_off * harev, xwexeA * harwn, xalphaA / a0, deA * harev
        write (f_out, 235) x0A * a0, xomegA * harwn, v0A * harev,
     &    v0A_off * harev, xwexeA * harwn, xalphaA / a0, deA * harev
      endif

      if (xwexeB .eq. 0.0d00) then
240     format ('x0B =',f6.2,2x, 'xomegB =',f9.2,2x, 'v0B =',f6.3,
     &    2x, 'v0B_off =', f6.3)
        write (*, 240) x0B * a0, xomegB * harwn, v0B * harev,
     &    v0B_off * harev
        write (f_out, 240) x0B * a0, xomegB * harwn, v0B * harev,
     &    v0B_off * harev
      else
245     format ('x0B =',f6.2,2x, 'xomegB =',f9.2,2x, 'v0B =',f6.3,
     &    2x, 'v0B_off =', f6.3,
     &    /, 2x,'wexeB =',f9.2,2x,'alphaB =',f9.3,2x,'DeB =',f9.3)
        write (*, 245) x0B * a0, xomegB * harwn, v0B * harev,
     &    v0B_off * harev, xwexeB * harwn, xalphaB / a0, deB * harev
        write (f_out, 245) x0B * a0, xomegB * harwn, v0B * harev,
     &    v0B_off * harev, xwexeB * harwn, xalphaB / a0, deB * harev
      endif

      if (xwexeC .eq. 0.0d00) then
250     format ('x0C =',f6.2,2x, 'xomegC =',f9.2,2x, 'v0C =',
     &    f16.8, 2x, 'v0C_off =', f6.3)
        write (*, 250) x0C * a0, xomegC * harwn, v0C * harev,
     &    v0C_off * harev
        write (f_out, 250) x0C * a0, xomegC * harwn, v0C * harev,
     &    v0C_off * harev
      else
255     format ('x0C =',f6.2,2x, 'xomegC =',f9.2,2x, 'v0C =',
     &    f16.8, 2x, 'v0C_off =', f6.3,
     &    /, 2x,'wexeC =',f9.2,2x,'alphaC =',f9.3,2x,'DeC =',f9.3)
        write (*, 255) x0C * a0, xomegC * harwn, v0C * harev,
     &    v0C_off * harev, xwexeC * harwn, xalphaC / a0, deC * harev
        write (f_out, 255) x0C * a0, xomegC * harwn, v0C * harev,
     &    v0C_off * harev, xwexeC * harwn, xalphaC / a0, deC * harev
      endif

430   format ('Shelf (eV) = ', f8.3)
      write (*, 430) shelf * harev
      write (f_out, 430) shelf * harev

C Calculate energy shift to apply in Fourier transform (depends on iCpot):

      if (iCpot .eq. 1) then
        e_shift = v0C_off - v0A_off
      else
        e_shift = v0C_off - v0B_off
      endif

290   format ('Energy shift passed to matrix.out =', f13.6)
      write (*, 290) e_shift * harev
      write (f_out, 290) e_shift * harev

C Calculate save interval for coverage of full energy range:

190   format('Energy range of FPES (eV) = ', f16.8)
200   format('Step size needed for full FT (fs) = ', f16.8)
      write (*, 190) harev * (vdiff_max - vdiff_min)
      write (*, 200) twopi * atu / (vdiff_max - vdiff_min)
      write (f_out, 190) harev * (vdiff_max - vdiff_min)
      write (f_out, 200) twopi * atu / (vdiff_max - vdiff_min)

C Calculate how often must propagate on upper surface to get wide enough FT:

      if (tCstep .eq. 0) then
        tCstep = tCfactor * twopi / (vdiff_max - vdiff_min)
400     format ('(tCstep automatic)')
        write (*, 400)
        write (f_out, 400)
      endif

C Recalculate tCstep based on itCstep (must be integer multiple of tstep):

      itCstep = int(tCstep / tstep) ! Use int to ensure is small enough (nint
C       rounds to nearest integer).
      tCstep = itCstep * tstep

210   format('Revised step size used for C surface (fs) = ',
     &  f10.4)
      write (*, 210) tCstep * atu
      write (f_out, 210) tCstep * atu

C Calculate number of wavefunctions to store for 2D autocorrelation, terminate
C if larger than max:

      num_queue = nint(proberange / tCstep)
460   format('num_queue = ', i4, ' MAX_QUEUE = ', i4)
      write (*, 460) num_queue, MAX_QUEUE
      write (f_out, 460) num_queue, MAX_QUEUE
      if (num_queue .gt. MAX_QUEUE) then
470     format('Error condition. Increase MAX_QUEUE in source code.',
     &    ' Terminating.')
        write (*, 470)
        write (f_out, 470)
        stop
      endif

C Print table of letters and corresponding times.

      if ((isavpsiA .eq. 1) .or. (isavpsiB .eq. 1)) then

        write (*, *)
        write (f_out, *)

445     format ('Table of delay times for saved wavefunctions.')
440     format ('File number  Delay/fs')
        write (*, 445)
        write (f_out, 445)
        write (*, 440)
```

```
        write (f_out, 440)

        it = nint(tmin / tstep) + 1
        t = tmin + tstep
450     format (4x, i3, 6x, f8.2)
        write (*, 450) 0, t * atu
        write (f_out, 450) 0, t * atu

4       if (mod (it, itpktsav) .ne. 0) then ! Find second save point
          it = it + 1
          t = t + tstep
          goto 4
        endif

        npacket = 1 ! Main loop
2         write (*, 450) npacket, t * atu
          write (f_out, 450) npacket, t * atu
          npacket = npacket + 1
          it = it + itpktsav
          t = t + itpktsav * tstep
        if (t .le. tmax) goto 2
      endif

C Generate absorbing boundary functions and save to files.

      call make_abs

472   format('Surface ', i1, ' iabs=',i1,' absfac=',f16.8, ' nxabs=',i4)
      write (f_out, 472) 1, iabsA, absfacA, nxabsA
      write (f_out, 472) 2, iabsB, absfacB, nxabsB
      write (*, 472) 1, iabsA, absfacA, nxabsA
      write (*, 472) 2, iabsB, absfacB, nxabsB

      if (iabsA .eq. 1) then
        open(f_temp, file='absA.out')
        do i = 1, nxabsA
471       format(f16.8)
          write(f_temp, 471) absA(i)
        enddo
        close(f_temp)
      endif
      if (iabsB .eq. 1) then
        open(f_temp, file='absB.out')
        do i = 1, nxabsB
          write(f_temp, 471) absB(i)
        enddo
        close(f_temp)
      endif

C ---------------------------------------------------------------------
C MAIN PROPAGATION CODE
C ---------------------------------------------------------------------

      zfactor = -2d0 * zeye * tstep / hb
      npacket = 0

C Open other files for run.

      open (f_normA, file = 'normA.out')
      open (f_normB, file = 'normB.out')
      open (f_matrix, file = 'matrix.out')

C Write vital information to overlap matrix file (including version header).

      write (f_matrix, *) ivers_matrix
      write (f_matrix, *) num_queue
      write (f_matrix, *) tmin * atu, tCstep * atu
      write (f_matrix, *) e_shift * harev

C Check whether to read wavepacket A from file, or generate from scratch.

      if (ireadA .eq. 1) then
        call psiread(1, zpsiA0)
      else
        call initA (tmin, zpsiA0)
      endif

C Generate zpsiA1 by second order Runge Kutta. This step is required to
C evaluate the time derivative in 2nd order differencing later on.

      call rk2 (1, zpsiA0, zpsiA1, zfactor) ! was tmin

C Check whether to read wavepacket B from file, or start as 0. Reverse momentum
C of wavepacket if requested.

      if (ireadB .eq. 1) then
        call psiread(2, zpsiB0)

        if (irevB .eq. 1) then
          do ix = 1, nXpts
            zpsiB0(ix) = dconjg(zpsiB0(ix))
          enddo
        endif

        call rk2 (2, zpsiB0, zpsiB1, zfactor) ! was tmin
      else
        do ix = 1, nXpts
          zpsiB0(ix) = zero
          zpsiB1(ix) = zero
        enddo
      endif

C Write initial wavepackets to disk (if surface is selected). Save initial norm.

      if (isavpsiA .eq. 1) then
        call pktsav (zpsiA0, npacket, 1)
      endif
      if (isavpsiB .eq. 1) then
        call pktsav (zpsiB0, npacket, 2)
      endif
      npacket = npacket + 1

C Set banner for subsequent output information.

      write (*, *)
      write (f_out, *)

360   format ('Time/fs    Norm A  Norm B  Norm C   KE A/eV  KE B',
     &  '/eV  KE C/eV  Steps')
      write (*, 360)
      write (f_out, 360)

C Begin t/it (inner) loop. t is time in atu; it is an integer counter (starts at
C some negative value) used for determining when to save wavepackets. nint =
C nearest integer. icount keeps track of wavefunction labels saved to disk.
```

```
      t = tmin + tstep
      it = nint (tmin / tstep) + 1
      iautocount = 1
      icount = 0
      i_queue = 0 ! Position of last wavepacket in zpsiqueue
      i_queue_flag = 0 ! Looparound flag initially not set

C Main loop start. Save wavepackets if it's time.

3     if (mod (it, itpktsav) .eq. 0) then
        if (isavpsiA .eq. 1) then
          call pktsav (zpsiA1, npacket, 1)
        endif
        if (isavpsiB .eq. 1) then
          call pktsav (zpsiB1, npacket, 2)
        endif
        npacket = npacket + 1
      endif

C zpsiA, zpsiB propagation: both optional.

      ZE = zfactor * ZE1(t)

      if (ipropA .eq. 1) then
        call Hpsi(1, t, zpsiA1, zHpsiA1)
        if (ipropB .eq. 1) then
          call Hpsi(2, t, zpsiB1, zHpsiB1)
          do ix = 1, nXpts
            zpsiAtemp = zpsiA0(ix) + ZE * zpsiB1(ix) +
     &        zfactor * zHpsiA1(ix) ! 6-30-97: shouldn't one be - ?
            zpsiBtemp = zpsiB0(ix) + ZE * zpsiA1(ix) +
     &        zfactor * zHpsiB1(ix)
            zpsiA0(ix) = zpsiA1(ix)
            zpsiB0(ix) = zpsiB1(ix)
            zpsiA1(ix) = zpsiAtemp
            zpsiB1(ix) = zpsiBtemp
          enddo
        else
          do ix = 1, nXpts
            zpsiAtemp = zpsiA0(ix) + ZE * zpsiB1(ix) +
     &        zfactor * zHpsiA1(ix) ! 6-30-97: shouldn't one be - ?
            zpsiA0(ix) = zpsiA1(ix)
            zpsiA1(ix) = zpsiAtemp
          enddo
        endif
      else
        if (ipropB .eq. 1) then
          call Hpsi(2, t, zpsiB1, zHpsiB1)
          do ix = 1, nXpts
            zpsiBtemp = zpsiB0(ix) + ZE * zpsiA1(ix) +
     &        zfactor * zHpsiB1(ix)
            zpsiB0(ix) = zpsiB1(ix)
            zpsiB1(ix) = zpsiBtemp
          enddo
        endif
      endif

C Apply absorbing boundary functions to A, B

      if (iabsA .eq. 1) then
        do i = 1, nxabsA
          j = nXpts - nxabsA + i
          zpsiA1(j) = zpsiA1(j) * absA(i)
        enddo
      endif
      if (iabsB .eq. 1) then
        do i = 1, nxabsB
          j = nXpts - nxabsB + i
          zpsiB1(j) = zpsiB1(j) * absB(i)
        enddo
      endif

C ZpsiC propagation.

      if ((ipropC .eq. 1) .and. (mod(it, itCstep) .eq. 0)) then

C   Copy appropriate surface wavefunction into zpsiC; generate C1 (note: back-
C   ward propagation!)

        if (iCpot .eq. 1) then
          do ix = 1, nXpts
            zpsiC0(ix) = zpsiA1(ix)
          enddo
        else
          do ix = 1, nXpts
            zpsiC0(ix) = zpsiB1(ix)
          enddo
        endif
        call rk2 (3, zpsiC0, zpsiC1, -zfactor) ! was tmin

C   Copy wavefunction into conjugated queue array.

        i_queue = i_queue + 1
        if (i_queue .gt. num_queue) then
          i_queue = 1
          i_queue_flag = 1 ! Set looparound flag
        endif
        do ix = 1, nXpts
          zpsiqueue(ix, i_queue) = dconjg(zpsiC0(ix))
        enddo

C   Propagate on C surface, recording overlap matrix element every itCstep
C   iterations.

        if ((isavpsiC .eq. 0) .or. (it .eq. 0)) then

          call chknrm(zpsiC0, rnormC)
          write (f_matrix, *) dcmplx(rnormC) ! Self-overlap

          if (i_queue_flag .eq. 1) then
            j_max = num_queue - 1
          else
            j_max = i_queue - 1
          endif
          nsteps = j_max * itCstep
          if (j_max .gt. 0) then
            do j = 1, j_max ! num_queue - 1
              i = i_queue - j
c             if ((i .le. 0) .and. (i_queue_flag .eq. 1)) then
              if (i .le. 0) then
                i = i + num_queue
              endif
c             if (i .gt. 0) then

              if (j .gt. 1) then ! OLD: if (j + 1 .ne. i_queue)
                kmax = itCstep
```

```
            else
              kmax = itCstep - 1 ! First step already done by rk2.
            endif
            do k = 1, kmax ! Propagate backward several times in succession.
C             Eventually this step may be replaced by more efficient Chebychev
C             routine.
              call Hpsi(3, t, zpsiC1, zHpsi)
              do ix = 1, nXpts
                z = zpsiC0(ix) - zfactor * zHpsi(ix)
                zpsiC0(ix) = zpsiC1(ix)
                zpsiC1(ix) = z
              enddo
            enddo

            do ix = 1, nXpts
              zovlp(ix) = zpsiC1(ix) * zpsiqueue(ix, i)
            enddo
            call zsimpint(nXpts, zovlp, dx, zaccum)
            write (f_matrix, *) zaccum ! was dconjg(zaccum)
c         endif

C DEBUG: Write intermed wf to disk if isavpsiC = 1 and it = 0:

            if ((isavpsiC .eq. 1) .and. (mod(j, 8) .eq. 0) .and.
     &        (it .eq. 0)) then
              write(*, *) 'Writing zpsiC1'
              write(f_out, *) 'Writing zpsiC1'
              call pktsav(zpsiC1, j, 3)
            endif

          enddo
        endif
      endif

C DEBUG: Terminate if isavpsiC = 1:

        if ((isavpsiC .eq. 1) .and. (it .eq. 0)) goto 800

C Done C surface propagation.

      endif

C Print statistics, write data to norm files.

      if (mod (it, istatstep) .eq. 0) then
        call chk (1, 1, zpsiA1, rnormA, HavA)
        call chk (2, 1, zpsiB1, rnormB, HavB)
900     format(f9.2, 2x, f6.4, 2x, f6.4, 2x, f6.4, 2x, f7.4, 2x,
     &    f7.4, 2x, f7.4, 2x, i5)
        write (*, 900) t * atu, rnormA, rnormB, rnormC, HavA *
     &    harev / rnormA, HavB * harev / rnormB, HavC * harev /
     &    rnormC, nsteps
        write (f_out, 900) t * atu, rnormA, rnormB, rnormC, HavA *
     &    harev / rnormA, HavB * harev / rnormB, HavC * harev /
     &    rnormC, nsteps

901     format(f9.2, 2x, f6.4)
        write (f_normA, 901) t * atu, rnormA
        write (f_normB, 901) t * atu, rnormB
      endif

C Autocorrelation chores.

      if (ieigenA .eq. 1) then

C Save initial wavepacket for autocorrelation when we reach starting time.

        if (it .eq. itautomin) then
          zautoA(1) = rnormA
          do ix = 1, nXpts
            zpsiAorig(ix) = zpsiA1(ix)
          enddo

C For subsequent values, perform autocorrelation.

        else if ((it .gt. itautomin) .and. (mod(it, itauto) .eq. 0))
     &    then
          call ovlp(zpsiAorig, zpsiA1, zaccum)
          iautocount = iautocount + 1
          zautoA(iautocount) = zaccum
        endif

C Check that we aren't blowing up.

        if ((rnormA .gt. 2d0) .or. (rnormB .gt. 2d0)) then
877       format ('Exceeded reasonable norm - terminating.')
          write (*, 877)
          write (f_out, 877)
          stop
        endif

      endif

C End t/it loop.

      t = t + tstep
      it = it + 1
      if (t .le. tmax) goto 3

C Save final wavepackets on each surface to a file.

      call psidump(1, zpsiA1)
      call psidump(2, zpsiB1)

C Perform FT of autocorrelation array. First must pad remaining terms with 0's.

      if (ieigenA .eq. 1) then
541     format('Performing FFT of autocorrelation. ',
     &    'Results saved in eigenA.out.')
        write (f_out, 541)
        write (*, 541)
        do i = iautocount + 1, nauto
          zautoA(i) = zero
        enddo

C DEBUG: Save original autocorrelation -- as both real part, and mag.

        open(f_temp, file='autoA.real')
        do i = 1, nauto
          write(f_temp, *) dreal(zautoA(i))
        enddo
        close(f_temp)

        open(f_temp, file='autoA.mag')
        do i = 1, nauto
          re = dreal(zautoA(i))
```

```
          ai = dimag(zautoA(i))
          write(f_temp, *) dsqrt(re * re + ai * ai)
        enddo
        close(f_temp)

C Multiply by smoothing function before taking FFT:

        do i = 1, nauto
          zautoA(i) = zautoA(i) * dcmplx(dexp(gamm * i * i))
        enddo

C DEBUG: Save smoothed autocorrelation -- as both real part, and mag.

        open(f_temp, file='autoA.sm.real')
        do i = 1, nauto
          write(f_temp, *) dreal(zautoA(i))
        enddo
        close(f_temp)

        open(f_temp, file='autoA.sm.mag')
        do i = 1, nauto
          re = dreal(zautoA(i))
          ai = dimag(zautoA(i))
          write(f_temp, *) dsqrt(re * re + ai * ai)
        enddo
        close(f_temp)

C Take FFT.

        call FFT(zautoA, nauto, 1)

C Write results to file - only write positive part since negative freq's don't
C have physical meaning.

        open(f_temp, file = 'eigenA.out')
        estep = twopi / nauto / tauto * harev
501     format(f9.5, 2x, e16.8)
        do i = 1, nauto / 2
          write(f_temp, 501) estep * (i - 1),
     &      dreal(zautoA(i))
        enddo
        close(f_temp)
      endif

C Now take final wavefunction and overlap with known eigenfunctions of X state.

      call ovlp(zpsiA1, zpsiA1, zaccum) ! Get norm of wavefunction
      atemp = dsqrt(dreal(zaccum))
      open(f_temp, file = 'eigenlist.out')
600   format(i2, 2x, e16.8)
      do nlevel = 0, 10
        call initA(0, zpsiA0) ! Use zpsiA0 as temp array to hold each e'fn.
        call ovlp(zpsiA1, zpsiA0, zaccum)
        re = dreal(zaccum / atemp)
        ai = dimag(zaccum / atemp)
       write(f_temp, 600) nlevel, re * re + ai * ai ! Must square result anyway.
      enddo
      close(f_temp)

C Done.

380   format('Done.')
800   write (*, 380)
      write (f_out, 380)

      close (f_out)
      close (f_normA)
      close (f_normB)
      close (f_matrix)

      stop
      end

C *********************************************************************************
      subroutine const ()
C *********************************************************************************
C Define some useful constants.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const1/ xmas, hb, sechfactor, gaussfactor

C Set conversion factors.

      harev = 27.211608d0
      evwn = 8065.479d0
      a0 = 0.52917706d0
      amu = 1822.882d0
      emu = 9.109534d-31
      harwn = harev * evwn
      amass = 1.66056d-27
      atu = 0.024199d0

C Set const0.

      zero = dcmplx (0.0d00, 0.0d00)
      zeye = dcmplx (0.0d00, 1.0d00)
      pi = dacos (-1.0d00)
      twopi = 2 * pi
      sqrtpi = dsqrt (pi)
      pisq = pi * pi
      alessthanhalf = 0.49999 ! Just a little less than half, but not so little
C       that we machine round to 0.5.

C Speed of light in cm/s and hbar in atomic units.

      c = 2.99792458d10
      hb = 1.0d0

C Pulse shape parameters: sechfactor is multiplied by time inside sech^2 so that
C relative pulse intensity = 0.5 when time = FWHM / 2. Gaussfactor is similar,
C but for a gaussian-shaped pulse.

      sechfactor = 1.762747174d00  ! = 2 * arccosh(sqrt(2))
      gaussfactor = -1.386294361d00  ! = -2 * ln(2)

      return
      end

C *********************************************************************************
      subroutine read_input ()
C *********************************************************************************
C Order of input deck:
```

```
C ivers: Version number of program. If different from current, terminates.

C isavpotA, isavpotB, isavpotC, isavpot_shift: Flag (0, 1) to save potential
C   functions of surfaces A, B, C to file. isavpot_shift is flag (0,1) to save
C   shifted potentials, rather than unmodified ones (for debugging purposes).

C isavpsiA, isavpsiB, isavpsiC, tpktsav: Flag (0, 1) to save wavefunction on
C   surfaces A, B, C to file. Note if isavpsiC set, program ends after first
C   call to chebprop. tpktsav is period for recording (fs).

C ieigenA, tauto: Flag (0, 1) to perform eigenspace decomposition on psiA, time
C   interval (fs) for saving autocorrelation. Use tauto = 0 for automatic calcu-
C   lation (will use largest value possible without overrunning array). Depha-
C   sing constant (used to eliminate high-frequency ringing in Fourier transform
C   due to finite record length) is calculated automatically.

C xmas: Reduced mass of system (a.m.u.).

C tmin, tmax, tstep: Range of propagation, step size on A/B surfaces. If tstep
C   = 0, calculates automatically.

C ipropA, ipropB, ipropC: Flags (0, 1) to propagate on A, B, C surfaces.

C iCpot, tCstep, tCfactor: Surface (1=A, 2=B) to take FPES from; interval (fs)
C   for C surface transfer (controls energy range of FT; if 0, is calculated
C   automatically); safety factor for automatic timestep.

C statstep: Interval (fs) for saving statistics (norm, KE) to screen.

C emin, emax, factor: Parameters for Chebychev approximation: emin and emax are
C   energy range (eV); factor is number of extra terms in approximation (for
C   safety).

c shelf: Potential shelf for all surfaces (eV).

C nlevel, Aoffset: Initial vibrational level of ground state (must be Morse
C   potential); alternatively, can specify energy offset to shift surface A with
C   Aoffset (eV).

C ireadA, ireadB, irevB: Flags (0,1) to read initial wavefunctions A, B from
C   file instead (note: nlevel irrelevant if ireadA = 1), flag (0,1) to reverse
C   momentum of wavepacket B when initially read in (for simulating reflection
C   off solvent cage).

C xmin, xmax, nXpts: Spatial limits of calculation (Angstroms), number of points
C   for grid.

C hv1, fwhm1, dmu1, itype1: Photon
C   energy (eV), FWHM pulse width (fs), dipole moment (angstrom) and type (0 =
C   sech^2, 1 = gaussian) of pump pulse.

C fwhm2, thresh: Maximum anticipated FWHM of probe (fs), intensity threshold
C   (between 0 and 1) of probe pulse (used to minimize the size of the 2D auto-
C   correlation array).

C iabsA, absfacA, nxabsA: Absorbing boundary flag, magnitude factor, number of
C   grid points from edge for surface A.  If absfacA = 0, generate factor auto-
C   matically.

C iabsB, absfacB, nxabsB: Absorbing boundary flag, magnitude factor, number of
C   grid points from edge for surface B.  If absfacB = 0, generate factor auto-
C   matically.

C ipottypA: Potential type for surface A = anion ground state. See potread ()
C   for details.

C ***: This line varies with ipottypA; see potread() for details.

C ipottypB: Potential type for surface B = anion excited state.

C ***: This line varies with ipottypB.

C ipottypC: Potential type for surface C = neutral.

C ***: This line varies with ipottypC.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const1/ xmas, hb, sechfactor, gaussfactor
      common /const2/ x0A, xomegA, v0A, xwexeA, dea, xalphaA, x0B,
     &  xomegB, xwexeB, deB, xalphaB, v0B, x0C, xomegC, xwexeC, deC,
     &  xalphaC, v0C, shelf, v0A_off, v0B_off, v0C_off
      common /const3/ tstep
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      common /const7/ tpktsav, itpktsav, tCstep, itCstep, tCfactor,
     &  emin, emax, factor, e_shift, tmin, tmax, statstep, istatstep,
     &  tauto, itauto
      common /const8/ hv1, fwhm1, dmu1, domega1, sech1, itype1, ipropA,
     &  ipropB, ipropC, iCpot, nlevel, Aoffset, Eint, fwhm2, thresh,
     &  proberange, ireadA, ireadB, irevB
      integer f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /files/ f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      character*80 input_header, last_date, matrix_header, s
      common /headers/ input_header, last_data, matrix_header
      common /pottyp/ ipottypA, ipottypB, ipottypC, isavpotA, isavpotB,
     &  isavpotC, isavpsiA, isavpsiB, isavpsiC, ieigenA, ieigenB,
     &  isavpot_shift
      common /version/ ivers_input, ivers_matrix


900   format('Reading alpine.inp.')
      write (*, 900)
      write (f_out, 900)

      open (f_in, file = 'alpine.inp')
      read (f_in, *) ivers
      if (ivers_input .ne. ivers) then
901     format('Input deck incompatible: ', i5, ' sought, ', i5,
     &    ' read.')
        write (*, 901) ivers_input, ivers
        write (f_out, 901) ivers_input, ivers
        stop
      endif

C Read in parameters

      read (f_in, *) isavpotA, isavpotB, isavpotC, isavpot_shift
      write(*,*) 'isavpotA=',isavpotA
```

```
      write(*,*) 'isavpotB=',isavpotB
      write(*,*) 'isavpotC=',isavpotC

      read (f_in, *) isavpsiA, isavpsiB, isavpsiC, tpktsav
      tpktsav = tpktsav / atu

      read (f_in, *) ieigenA, tauto
      tauto = tauto / atu

      read (f_in, *) xmas
      xmas = xmas * amu

      read (f_in, *) tmin, tmax, tstep
      tmin = tmin / atu
      tmax = tmax / atu
      tstep = tstep / atu

      read (f_in, *) ipropA, ipropB, ipropC

      read (f_in, *) iCpot, tCstep, tCfactor
      tCstep = tCstep / atu ! Deal with neg value later.

      read (f_in, *) statstep
      statstep = statstep / atu

      read (f_in, *) emin, emax, factor
      emin = emin / harev
      emax = emax / harev

      read (f_in, *) shelf
      shelf = shelf / harev

      read (f_in, *) nlevel, Aoffset
      Aoffset = Aoffset / harev

      read (f_in, *) ireadA, ireadB, irevB

      read (f_in, *) xmin, xmax, nXpts
      xmin = xmin / a0
      xmax = xmax / a0
      dx = (xmax - xmin) / nXpts

      read (f_in, *) hv1, fwhm1, dmu1, itype1
      hv1 = hv1 / harev
      fwhm1 = fwhm1 / atu
      dmu1 = dmu1 / a0
      domega1 = hv1 / hb
      if (itype1 .eq. 0) then
        sech1 = sechfactor / fwhm1
      else
        sech1 = gaussfactor / (fwhm1 * fwhm1)
      endif

      read (f_in, *) fwhm2, thresh
      fwhm2 = fwhm2 / atu
      proberange = fwhm2 * thresh

      read (f_in, *) iabsA, absfacA, nxabsA
      read (f_in, *) iabsB, absfacB, nxabsB

C Read in potential energy function parameters.

      call potread ()
```

```
C Done.

      close(f_in)

      return
      end

C *********************************************************************
      subroutine potread()
C *********************************************************************
C reads in potential energy function parameters, then computes (or reads from a
C file) the value of the function at each grid point.

C Potential types available are:
C 0:  Morse (input Re, De and Be)
C 1:  Harmonic/Anharmonic (input Re, We, Wexe). If wexe .ne. 0 then uses Morse.
C 3:  Potential read from file - special format

C Other potentials may be added by modifying the subroutine.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const1/ xmas, hb, sechfactor, gaussfactor
      common /const2/ x0A, xomegA, v0A, xwexeA, dea, xalphaA, x0B,
     &  xomegB, xwexeB, deB, xalphaB, v0B, x0C, xomegC, xwexeC, deC,
     &  xalphaC, v0C, shelf, v0A_off, v0B_off, v0C_off
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      common /pot/ zpot(npts, 3)
      common /pottyp/ ipottypA, ipottypB, ipottypC, isavpotA, isavpotB,
     &  isavpotC, isavpsiA, isavpsiB, isavpsiC, ieigenA, ieigenB,
     &  isavpot_shift
      integer f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /files/ f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3

C SURFACE A (anion ground state)-----------------------------------------------

C set the potential by reading potential type

      read(f_in,*)ipottypA

C For each type read relevant parameters:
C (expect x0 in Angs, omega, and wexe in cm-1 and Vo, De in eV, alpha in Angs-1)

      if (ipottypA .eq. 0) then
        read(f_in,*)x0A,deA,xalphaA,v0A
C convert to au
        x0A=x0A/a0
        deA=deA/harev
        xalphaA=xalphaA*a0
        xwexeA=xalphaA**2/(2.0*xmas)
        xomegA=dsqrt(4.0*xwexeA*deA)
        v0A=v0A/harev
        call morse(1,x0A,deA,xalphaA,v0A)

      else if (ipottypA .eq. 1) then
```

```
        read(f_in,*)x0A,xomegA,xwexeA,v0A
C convert to au
        x0A=x0A/a0
        xomegA=xomegA/harwn
        xwexeA=xwexeA/harwn
        v0A=v0A/harev
C If Morse, convert potential parameters to reciprocal bohr and hartrees...
        if (xwexeA .ne. 0.0d00) then
          xalphaA=dsqrt(2.0d00*xmas*xwexeA)
          deA=xomegA**2/(4.0d00*xwexeA)
c         ipottypA=0
          call morse(1,x0A,deA,xalphaA,v0A)
        else
          call harmonic(1,xmas,x0A,xomegA,v0A)
        endif

      else if (ipottypA .eq. 3) then
        v0A = read_user_pot(1)

      else
100     format('Problem with surface A potential type.')
        write (*, 100)
        write (f_out, 100)
        stop
      endif

C SURFACE B (anion excited state)-----------------------------------------------

      read(f_in,*)ipottypB

      if (ipottypB .eq. 0) then
        read(f_in,*)x0B,deB,xalphaB,v0B
        x0B=x0B/a0
        deB=deB/harev
        xalphaB=xalphaB*a0
        xwexeB=xalphaB**2/(2.0*xmas)
        xomegB=dsqrt(4.0*xwexeB*deB)
        v0B=v0B/harev
        call morse(2,x0B,deB,xalphaB,v0B)
      else if (ipottypB .eq. 1) then
        read(f_in,*)x0B,xomegB,xwexeB,v0B
        x0B=x0B/a0
        xomegB=xomegB/harwn
        xwexeB=xwexeB/harwn
        v0B=v0B/harev
        if (xwexeB .ne. 0.0d00) then
          xalphaB=dsqrt(2.0d00*xmas*xwexeB)
          deB=xomegB**2/(4.0d00*xwexeB)
          ipottypB=0
          call morse(2,x0B,deB,XalphaB,v0B)
        else
          call harmonic(2,xmas,x0B,xomegB,v0B)
        endif

      else if (ipottypB .eq. 3) then
        v0B = read_user_pot(2)

      else
110     format('Problem with surface B potential type.')
        write (*, 110)
        write (f_out, 110)
        stop
      endif
```

```
C SURFACE C (neutral)-----------------------------------------------------------

      read(f_in,*)ipottypC

      if (ipottypC .eq. 0) then
        read(f_in,*)x0C,deC,xalphaC,v0C
        x0C=x0C/a0
        deC=deC/harev
        xalphaC=xalphaC*a0
        xwexeC=xalphaC**2/(2.0*xmas)
        xomegC=dsqrt(4.0*xwexeC*deC)
        v0C=v0C/harev
        call morse(3,x0C,deC,xalphaC,v0C)

      else if (ipottypC .eq. 1) then
         read(f_in,*)x0C,xomegC,xwexeC,v0C
         x0C=x0C/a0
         xomegC=xomegC/harwn
         xwexeC=xwexeC/harwn
         v0C=v0C/harev
         if (xwexeC .ne. 0.0d00) then
           xalphaC=dsqrt(2.0d00*xmas*xwexeC)
           deC=xomegC**2/(4.0d00*xwexeC)
           ipottypC=0
           call morse(3,x0C,deC,XalphaC,v0C)
         else
           call harmonic(3,xmas,x0C,xomegC,v0C)
         endif

      else if (ipottypC .eq. 3) then
        v0C = read_user_pot(3)

      else
120     format('Problem with surface C potential type.')
        write (*, 120)
        write (f_out, 120)
        stop
      endif

      return
      end

C *********************************************************************************
      subroutine harmonic (ipot, xmas, x0, xomeg, v0)
C *********************************************************************************
C 12/8/96 (BJG) Note shelf moved to main subroutine.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const2/ x0A, xomegA, v0A, xwexeA, dea, xalphaA, x0B,
     &  xomegB, xwexeB, deB, xalphaB, v0B, x0C, xomegC, xwexeC, deC,
     &  xalphaC, v0C, shelf, v0A_off, v0B_off, v0C_off
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      common /pot/ zpot(npts, 3)

99    format (a, f5.1, a)
      write (6, *)

      do 120 ix = 1, nXpts
```

```
        xi = xmin + (ix - 1) * dx
        zpot(ix, ipot) = 0.50d00 * xmas * (xomeg * (xi - x0)) ** 2 + v0
120   continue

      return
      end

C *********************************************************************************
      subroutine morse (ipot, x0, de, xalpha, v0)
C *********************************************************************************
C 12/8/96 (BJG) Note shelf moved to main subroutine.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const2/ x0A, xomegA, v0A, xwexeA, dea, xalphaA, x0B,
     &  xomegB, xwexeB, deB, xalphaB, v0B, x0C, xomegC, xwexeC, deC,
     &  xalphaC, v0C, shelf, v0A_off, v0B_off, v0C_off
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      common /pot/ zpot(npts, 3)

99    format (a, f5.1, a)
      write (6, *)

      do 120 ix = 1, nXpts
        xi = xmin + (ix - 1) * dx
        zpot(ix, ipot) = de * (1.0d00 - dexp (-xalpha * (xi - x0))) ** 2
     &    + v0
120   continue

      return
      end

C *********************************************************************************
      real*8 function read_user_pot(ipot)
C *********************************************************************************
C Read v0, potential filename from input_deck, then read potential from file,
C save in zpot(ipot). Perform interpolation automatically.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      common /pot/ zpot(npts, 3)
      integer f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /files/ f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3

      character*80 pot_fn
      real*8 dx_temp, nXpts_temp, v0, vtemp(npts),
     &  x, xmin_temp, xmax_temp, x_temp
      integer ifile, ix, ix_temp

C Read potential

      read(f_in, *) v0
      if (ipot .eq. 1) then
        pot_fn = 'potA.in'
      else if (ipot .eq. 2) then
        pot_fn = 'potB.in'
      else if (ipot .eq. 3) then
        pot_fn = 'potC.in'
      endif
c     read(f_in, *) v0, pot_fn
      v0 = v0 / harev

100   format('Pot = ', i1, '  File = ', a)
      write(f_out, 100) ipot, pot_fn
      write(*, 100) ipot, pot_fn

      open(f_temp, file = pot_fn)
      read(f_temp, *) xmin_temp, xmax_temp, nXpts_temp
      xmin_temp = xmin_temp / a0
      xmax_temp = xmax_temp / a0
      dx_temp = (xmax_temp - xmin_temp) / nXpts_temp

      do ix = 1, nXpts_temp
        read(f_temp, *) vtemp(ix)
        vtemp(ix) = vtemp(ix) / harev + v0
      enddo
      close(f_temp)

C Interpolate potential to fit current grid

      do ix = 1, nXpts
        x = xmin + (ix - 1) * dx
        if (x .lt. xmin_temp) then
          zpot(ix, ipot) = vtemp(1)
        else if (x .ge. xmax_temp - dx_temp) then !990222 BJG: Added -dx_temp
          zpot(ix, ipot) = vtemp(nXpts_temp)
        else
          ix_temp = int((x - xmin_temp) / dx_temp) + 1
          x_temp = xmin_temp + (ix_temp - 1) * dx_temp
          zpot(ix, ipot) = (vtemp(ix_temp) * (x_temp +
     &      dx_temp - x) + vtemp(ix_temp + 1) * (x - x_temp))
     &      / dx_temp
        endif
      enddo

      read_user_pot = v0
      return
      end

C *********************************************************************************
      subroutine initA (ti, zpsiA)
C *********************************************************************************
C Initialize the wavefunction array on anion ground state surface.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const1/ xmas, hb, sechfactor, gaussfactor
      common /const2/ x0A, xomegA, v0A, xwexeA, dea, xalphaA, x0B,
     &  xomegB, xwexeB, deB, xalphaB, v0B, x0C, xomegC, xwexeC, deC,
     &  xalphaC, v0C, shelf, v0A_off, v0B_off, v0C_off
      common /const3/ tstep
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
```

```
      common /pottyp/ ipottypA, ipottypB, ipottypC, isavpotA, isavpotB,
     &  isavpotC, isavpsiA, isavpsiB, isavpsiC, ieigenA, ieigenB,
     &  isavpot_shift
      integer f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /files/ f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3

C Local declarations:

      dimension zpsiA (npts)

C Place the initial wavepacket on surface A.
C If using a fully flexible potential, i.e., from a file, then need the
C initial (ground) wavefunction of the ground state surface supplied
C explicitly: use anionwf subroutine that reads ld wavefunction from file
C in the format produced by the FCF program of Ellison.

      if (ipottypA .eq. 3) then
         call anionwf (1, zpsiA)
      else
      if (ipottypA .eq. 4) then
           call psiread (1, zpsiA)
      else
c     if (ipottypA .eq. 1) then
c       call initWF (zpsiA)
c     else
        call morsewf (zpsiA)
c     endif
      endif
      endif

C Check the norm and energy of the stationary state on the lower potential

      call chk (1, ti, zpsiA, rsnorm, Have)

      write (*, *)
      write (f_out, *)

100   format('Norm of initial wavefn is ', f12.6)
      write (*, 100) rsnorm
      write (f_out, 100) rsnorm

110   format('Energy (on lower surface) <H> =', f10.5, 2x, f10.2,
     &  'cm^-1')
      write (*, 110) Have/rsnorm, Have*harwn/rsnorm
      write (f_out, 110) Have/rsnorm, Have*harwn/rsnorm

      return
      end

C *******************************************************************************
      subroutine rk2 (wf, zpsiA0, zpsiA1, zfactor)
C     subroutine rk2 (wf, zpsiA0, zpsiA1, ti)
C *******************************************************************************
C Generate zpsiA1 from zpsiA0 using second-order Runge Kutta ("RK2") on surface
C wf (1-3).
C 8/13/97 note: Made more flexible by including zfactor as argument (so can
C propagate backward); more efficient by calculating fractions of zfactor once.
C Got rid of ti -- never used.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const1/ xmas, hb, sechfactor, gaussfactor
      common /const2/ x0A, xomegA, v0A, xwexeA, dea, xalphaA, x0B,
     &  xomegB, xwexeB, deB, xalphaB, v0B, x0C, xomegC, xwexeC, deC,
     &  xalphaC, v0C, shelf, v0A_off, v0B_off, v0C_off
      common /const3/ tstep
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB

C Local declarations:

      dimension zpsiA0(npts), zpsiA1(npts)
      dimension zHpsi(npts), zpsiAI(npts)
C     dimension zHpsiA0(npts), zpsiAI(npts), zHpsiAI(npts)
      complex*16 zfactor_div_4, zfactor_div_2

      zfactor_div_4 = zfactor / 4.
      zfactor_div_2 = zfactor / 2.

      call Hpsi(wf, ti, zpsiA0, zHpsi)
      do ix = 1, nXpts
        zpsiAI(ix) = zpsiA0(ix) + zfactor_div_4 * zHpsi(ix)
      enddo
      call Hpsi(wf, ti, zpsiAI, zHpsi)
      do ix = 1, nXpts
        zpsiA1(ix) = zpsiA0(ix) + zfactor_div_2 * zHpsi(ix)
      enddo

c     call Hpsi(wf, ti, zpsiA0, zHpsiA0)
c     do 20 ix = 1, nXpts
c       zpsiAI(ix) = zpsiA0(ix) - zeye * (tstep / 2.00d00) * zHpsiA0(ix)
c    &     / hb
c20    continue
c     call Hpsi(wf, ti, zpsiAI, zHpsiAI)
c     do 30 ix = 1, nXpts
c       zpsiA1(ix) = zpsiA0(ix) - zeye * tstep * zHpsiAI(ix) / hb
c30    continue

      return
      end

C *******************************************************************************
      subroutine initWF (zpsiA0)
C *******************************************************************************
C Initialize wavefunction on surface A.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const1/ xmas, hb, sechfactor, gaussfactor
      common /const2/ x0A, xomegA, v0A, xwexeA, dea, xalphaA, x0B,
     &  xomegB, xwexeB, deB, xalphaB, v0B, x0C, xomegC, xwexeC, deC,
     &  xalphaC, v0C, shelf, v0A_off, v0B_off, v0C_off
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB

C Local declarations:
```

```
      dimension zpsiA0(npts)

C Initial wavefn on surface is ground harmonic oscillator. Only does the ground
C state wavefunction (lowest quantum state); for higher vibrational wavefunc-
C tions see the 2-D code.

       if (xomegA .eq. 0.0d00) then
         write (6, *) 'No initial wavepacket as no omega available'
         stop
       endif
       xt = x0A
       pt = 0.0d00
       zat = dcmplx (0.0d00, xmas * xomegA / 2.0d00)
       gt = -(hb / 4.0d00) * dlog (2 * dimag (zat) / (pi * hb))
       zgt = dcmplx (0.0d00, gt)

C         write(2,910) xt,pt,zat,gt
910      format (5(1x,e13.6))

       do 10 ix = 1, nXpts
         xi = xmin + (ix - 1) * dx
         zarg = zat * (xi - xt) * (xi - xt) + pt * (xi - xt) + zgt
         zpsiA0(ix) = exp(zeye * zarg / hb)
10     continue

       return
       end

C *************************************************************************
      subroutine Hpsi (ipot, ti, zpsi, zHpsi)
C *************************************************************************
C Compute H * psi = ( KE + PE ) * psi(x).

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const1/ xmas, hb, sechfactor, gaussfactor
      common /const3/ tstep
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB

C Local declarations:

      dimension zpsi(npts), zHpsi(npts)
      dimension zpsiPE(npts),zpsiKE(npts)

      call KEmat (zpsi, zpsiKE, ti)
      call PEmat (ipot, zpsi, zpsiPE)

      do 10 ix = 1, nXpts
        zHpsi(ix) = zpsiPE(ix) + zpsiKE(ix)
10    continue

      return
      end

C *************************************************************************
       subroutine KEmat (zpsiX, zpsiK, t)
C *************************************************************************
C Computes (-hb ** 2) / (2 * xmas)) * (d / dx) ** 2[zpsiX] = zpsiK. Note
C zpsix(1) <-> zpsi(x0), zpsi(nXpts) <-> zpsix(xf), etc. Uses forward and back-
C ward FFT to evaluate 2nd derivative.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const1/ xmas, hb, sechfactor, gaussfactor
      common /const3/ tstep
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB

C Local declarations:

      dimension zpsiX(npts), zpsiK(npts)

C Backward Fourier transform : zpsiX(x) => zpsiK(k).

      isign = -1
      do 10 ix = 1, nXpts
10    zpsiK(ix) = zpsiX(ix)
      call FFT (zpsiK, nXpts, isign)

C Compute the second derivative in the momentum domain.

      L = nXpts / 2
      do 20 k = 0, nXpts - 1
        if (k .le. L) then
          zpsiK(k + 1) = -k * k * zpsiK(k + 1) / nXpts
        else
          zpsiK(k + 1) = -(nXpts - k) * (nXpts - k) * zpsiK(k + 1) /
     &      nXpts
        endif
20    continue

C Forward transform : zpsiK(k) => zpsix(x).

      isign = 1
      call FFT(zpsiK, nXpts, isign)

C Scale results.

      xL = xmax - xmin
      c1 = -0.5d00 * hb * hb / xmas
      c2 = 4 * pisq / (xL * xL)
      c = c1 * c2
      do 30 ix = 1, nXpts
        zpsiK(ix) = c * zpsiK(ix)
30    continue

      return
      end

C *************************************************************************
      subroutine PEmat (ipot, zpsiX, zpsiP)
C *************************************************************************
C Calculate zpot(x) * zpsiX = zpsiP. Note zpsiX(1) <-> zpsiX(x0), zpsiX(nxpts)
C <-> zpsiX(xf), etc.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
```

```
     &  iabsB,nxabsB,absfacB
      common /pot/ zpot(npts, 3)

C Local declarations:

      dimension zpsiX(npts), zpsiP(npts)

      do 10 ix = 1, nXpts
        zpsiP(ix) = zpot(ix, ipot) * zpsiX(ix)
10    continue

      return
      end

C *************************************************************************
      subroutine potlsave(ipot)
C *************************************************************************
C Save potential function ipot (= 1-3) to file.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      common /pot/ zpot(npts, 3)
      integer f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /files/ f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3

      if (ipot .eq. 1) open (f_temp, file = 'potA.out')
      if (ipot. eq. 2) open (f_temp, file = 'potB.out')
      if (ipot. eq. 3) open (f_temp, file = 'potC.out')
      do 10 ix = 1, nXpts
        xi = a0 * (xmin + (ix - 1) * dx)
        a = dreal (zpot(ix, ipot)) * harev
        write (f_temp, 930) xi, a
930     format (2x, f8.3, 2x, f20.10)
10    continue

      close (f_temp)
      return
      end

C *************************************************************************
      subroutine FFT_prep
C *************************************************************************
C Create array of values to be used by FFT routine; must call this before
C any calls to FFT.

      implicit real*8 (A-H, O-Y)
      parameter (npts=2048)
      complex*16 s, cstore(npts)
      common /fftvars/ cstore

C The roots of unity exp(pi*i*k/j) for j=1,2,4,..,n/2 and k=0,1,2,..,j-1
C are computed once and stored in a table. This table is used in subse-
C quent calls of fft.

      pi = 3.14159265358979d00
      j = 1
      icnt = 0
10    s = pi * (0, 1) / j
      do 20 k = 0, j - 1
        icnt = icnt + 1
20    cstore(icnt) = exp(s * k)
      j = j + j
      if (j .lt. npts) goto 10

      return
      end

C *************************************************************************
      subroutine FFT (x, n, isign)
C *************************************************************************

C ***************************************************************
C * The fft computes the discrete fast Fourier transform of a   *
C * sequence of n terms.                                        *
C * The forward FFT computes                                    *
C *         y(j)= sum (from k=0 to n-1) x(k)*exp(2*pi*i*j*k/n)  *
C * the backward FFT computes                                   *
C *         y(j)= sum (from k=0 to n-1) x(k)*exp(-2*pi*i*j*k/n) *
C *                                                             *
C *  x is a complex array of length n.                          *
C *  n is a power of 2.  n<=16384                               *
C * isign is the direction of the transform.  If isign >= 0 then*
C * the fft is forward , otherwise backward.                    *
C *                                                             *
C *  Ref.  Cooley, Lewis, Welch.  The FFT and its applications  *
C *         IEEE Trans. on Education, vol. E-12 #1; p. 29       *
C ***************************************************************

C 1997.10.3 BJG: Note that original implementation, where first call
C automatically generates an array of values for use in subsequent calls,
C has been moved to a separate function FFT_prep which must be called
C first!

      implicit real*8 (A-H, O-Y)
      parameter (npts=2048)
      complex*16 v, w, x(npts), cstore(npts)
      common /fftvars/ cstore

C  ******Bit reversal**********
C The x(j) are permuted in such a way that each new place number j is
C the bit reverse of the original placenumber.

      j = 1
      do 30 i = 1, n
        if (i .le. j) then
          v = x(j)
          x(j) = x(i)
          x(i) = v
        endif
        m = n / 2
25      continue
        if (j .gt. m) then
          j = j - m
          m = m / 2
          if (m .ge. 1) go to 25
        else
          j = j + m
        endif
30    continue
```

```
C ***********Matrix multiplication**************
C The roots of unity and the x(j) are multiplied.

      j = 1
      icnt = 0
40    jj = j + j
      do 50 k = 1, j
        icnt = icnt + 1
        w = cstore(icnt)
        if (isign .lt. 0) w = dconjg (w)
        do 50 i = k, n, jj
          v = w * x(i + j)
          x(i + j) = x(i) - v
50        x(i) = x(i) + v
      j = jj
      if (j .lt. n) goto 40

      return
      end

C *************************************************************************
      subroutine chk (ipot, ti, zpsiA, rnorm, Hav)
C *************************************************************************
C Check that norm and energy are conserved.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)

C Local declarations:

      dimension zpsiA(npts)

      call chknrm (zpsiA, rnorm)
      call chken (ipot, ti, zpsiA, Hav)

      return
      end

C *************************************************************************
      subroutine chknrm (zpsi, rnorm)
C *************************************************************************
C Check that the norm is conserved during numerical integration of TDSE.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB

C Local declarations:

      dimension zpsi(npts), psisq(npts)

      do 10 ix = 1, nXpts
        rpsi = dreal (zpsi(ix))
        aipsi = dimag (zpsi(ix))
        psisq(ix) = rpsi * rpsi + aipsi * aipsi
10    continue

      call simpint (nXpts, psisq, dx, rnorm)

      return
      end

C *************************************************************************
      subroutine chken (ipot, ti, zpsiA, Hav)
C *************************************************************************
C Check that energy is conserved during numerical intergration of the TDSE.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB

C Local declarations:

      dimension zpsiA(npts), zHpsiA(npts), psiHpsi(npts)

      call Hpsi (ipot, ti, zpsiA, zHpsiA)
      do 10 ix = 1, nXpts
        psiHpsi(ix) = dreal(dconjg(zpsiA(ix)) * zHpsiA(ix))
10    continue

      call simpint (nXpts, psiHpsi, dx, Hav)

      return
      end

C *************************************************************************
      subroutine simpint(nx, f1, dx, fint)
C *************************************************************************
C Simpson Rule integrator. This subprogram calls the trapezoidal integrator
C twice. Because of cancellation of errors the result is accurate to the the
C order of (1/nx**4). Rule valid only when nx odd. Hence for even nx the last
C piece of area under f1(nx-1) and f2(nx) is added by trapezoidal rule.
C
C Reference 'Numerical recipes'  Press, Flannery, Teukolsky, Vetterling
C Cambridge University Press, Cambridge (1986).

      implicit real*8 (A-H, O-Y)

C Local declarations:

      parameter (nypts = 2048)
      dimension f1(nypts), f2(nypts)

C Define:

      dx1 = dx
      dx2 = 2.0d00 * dx
      ixn = 0

      if (nx .gt. nypts) then
        write (6, *) ' simpint : nx .gt. nxpts = ', nypts
      endif

      if (mod (nx, 2) .eq. 0) then
        nx1 = nx - 1
        nx2 = 0.50d00 * nx1 + 1
        fint = 0.50d00 * dx * (f1(nx - 1)  + f1(nx))
      else
        nx1 = nx
        nx2 = 0.50d00 * nx1 + 1
        fint = 0.0d00
```

```
      endif

C Copy the odd elements of farray into f2.

      do 10 ix = 1, nx1, 2
        ixn = ixn + 1
10      f2(ixn) = f1(ix)

C Now integrate f1, f2 in two pieces.

      call trapint (nx1, f1, dx1, fint1)
      call trapint (nx2, f2, dx2, fint2)
      fint = fint + (4.0d00 * fint1 - fint2) / 3.0d00

      return
      end

C ********************************************************************************
      subroutine trapint (npts, f, dx, fint)
C ********************************************************************************

      implicit real*8 (A-H, O-Y)

C Local declarations:

      dimension f(npts)

C Trapeziodal rule integrator for f(1) - f(npts) <-> f(x0) - f(xf).

      fint = 0.

      do 100 i = 2, npts - 1
        fint = fint + f(i)
100   continue

      fint = fint + (f(1) + f(npts)) / 2.0d00
      fint = fint * dx

      return
      end

C ********************************************************************************
      subroutine ovlp(zpsi1, zpsi2, zovp)
C ********************************************************************************
C Find the overlap integral.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB

C Local declarations:

      dimension zpsi1(npts), zpsi2(npts), zprod(npts)

      do 10 ix = 1, nXpts
        zprod(ix) = dconjg(zpsi1(ix)) * zpsi2(ix)
10    continue

      call zsimpint(nXpts, zprod, dx, zovp)

      return
      end

C ********************************************************************************
      subroutine zsimpint(nx, zf1, dx, zint)
C ********************************************************************************
C Complex Simpson's rule integrator.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)

C Local declarations:

      parameter (nypts = 2048)
      dimension zf1(nypts), zf2(nypts)

C Define:

      dx1 = dx
      dx2 = dx * 2.
      ixn = 0.

      if (nx .gt. nypts) then
        write (6,*) ' zsimpint : nx .gt. nypts = ', nypts
      endif

      if ((mod(nx, 2) .eq. 0)) then
        nx1 = nx - 1
        nx2 = 0.50d00 * nx1 + 1
        zint = 0.50d00 * dx * (zf1(nx - 1) + zf1(nx))
      else
        nx1 = nx
        nx2 = 0.50d00 * nx1 + 1
        zint = 0.0d00
      endif

C Copy the odd elements of zf1 array into zf2.

      do 10 ix = 1, nx1, 2
        ixn = ixn + 1
10      zf2(ixn) = zf1(ix)

C Now integrate zf1, zf2 in two pieces.

      call ztrapint(nx1, zf1, dx1, zint1)
      call ztrapint(nx2, zf2, dx2, zint2)
      zint = zint + (4.0d00 * zint1 - zint2) / 3.0d00

      return
      end

C ********************************************************************************
      subroutine ztrapint(npts, zf, dx, zint)
C ********************************************************************************

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf

C Local declarations:

      dimension zf(npts)
```

```
C Trapezoidal rule integrator for f(1) - f(npts) <-> f(x0) - f(xf).

      zint = zero

      do 100 i = 2, npts - 1
        zint = zint + zf(i)
100   continue
      zint = zint + (zf(1) + zf(npts)) / 2.0d00
      zint = zint * dx

      return
      end

C *************************************************************************
      function ZE1(t)
C *************************************************************************
C Returns (complex) value of dipole moment AB * E-field of laser pulse 1 at time
C t. Note that cosh term is not squared; I = E^2 = 1/cosh^2, but this is calcu-
C lating just E = 1/cosh.

C 8dec96: Modified to not use frequency since is always set to 0 by shifting
C potentials around.

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common /const8/ hv1, fwhm1, dmu1, domega1, sech1, itype1, ipropA,
     &  ipropB, ipropC, iCpot, nlevel, Aoffset, Eint, fwhm2, thresh,
     &  proberange, ireadA, ireadB, irevB

      if (itype1 .eq. 0) then
        ZE1 = dmu1 / cosh (sech1 * t)
      else
        ZE1 = dmu1 * dexp (sech1 * t * t)
      endif

c     if (itype1 .eq. 0) then
c       ZE1 = dmu1 * cexp(cmplx(-zeye * domega1 * t)) / cosh (sech1 * t)
c     else
c       ZE1 = dmu1 * cexp(cmplx(-zeye * domega1 * t)) * dexp(sech1 * t *
c    &    t)
c     endif

      return
      end

C *************************************************************
      subroutine pktsav (zpsi, npkt,ipot)
C *************************************************************
C Save wavepacket in different file for each shot and each potential surface
C selected. Code is psiXY.Z, where:
C
C   X = surface A, B, C, indicated by ipot = 1, 2, 3.
C   Y = A, B, etc. for first, second, etc. save interval.
C   Z = sq (position and square)
C       real (position and real part)
C       dump (real and imaginary part, useful for reading in later).
C       mom (position and square, in momentum representation) (only if enabled)
C
C Also append current wavepacket to wavepktX.out (X = surface A, B, C).

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq,
     &  alessthanhalf
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      integer f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /files/ f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3

C Local declarations:

      dimension zpsi(npts), zpsik(npts)

C First do position representations:

      open(f_temp, file = 'psi' // char(64 + ipot) //
     &  char(48 + npkt / 100) // char(48 + mod(npkt / 10, 10))
     &  // char(48 + mod(npkt, 10)) // '.sq')
      open(f_temp2, file = 'psi' // char(64 + ipot) //
     &  char(48 + npkt / 100) // char(48 + mod(npkt / 10, 10))
     &  // char(48 + mod(npkt, 10)) // '.real')
      open(f_temp3, file = 'psi' // char(64 + ipot) //
     &  char(48 + npkt / 100) // char(48 + mod(npkt / 10, 10))
     &  // char(48 + mod(npkt, 10)) // '.dump')
c       open (f_temp, file = 'psi' // char (64 + ipot) // char (65 +
c     &  npkt) // '.sq')
c       open (f_temp2, file = 'psi' // char (64 + ipot) // char (65 +
c     &  npkt) // '.real')
c       open (f_temp3, file = 'psi' // char (64 + ipot) // char (65 +
c     &  npkt) // '.dump')

      do ix = 1, nXpts
        xi = xmin + (ix - 1) * dx
        xiA = xi * a0
        re = dreal (zpsi(ix))
        ai = dimag (zpsi(ix))
        sq = re * re + ai * ai
902     format (2x, f8.3, 2x, f20.10)
          write (f_temp, 902) xiA, sq
          write (f_temp2, 902) xiA, re
          write (f_temp3, *) zpsi(ix)
      enddo
      close (f_temp)
      close (f_temp2)
      close (f_temp3)

C Now do momentum representation:

      do ix = 1, nXpts
        zpsik(ix) = zpsi(ix)
      enddo

      call FFT (zpsik, nXpts, -1) ! Backward Fourier transform
      call chknrm(zpsik, rnorm) ! Get normalization constant
      if (rnorm .eq. 0) then ! Trap for /0 error
        rnorm = 1
      endif

      open(f_temp, file = 'psi' // char(64 + ipot) //
```

```
     &  char(48 + npkt / 100) // char(48 + mod(npkt / 10, 10))
     &  // char(48 + mod(npkt, 10)) // '.mom')
c      open (f_temp, file = 'psi' // char (64 + ipot) // char (65 +
c     &  npkt) // '.mom')

       L = nXpts / 2
       xL = xmax - xmin
       c2 = twopi / xL

      do i = -L, L - 1
        if (i .lt. 0) then
          ix = nXpts + i + 1
        else
          ix = i + 1
        endif
        xiA = c2 * i
        re = dreal (zpsik(ix))
        ai = dimag (zpsik(ix))
        sq = re * re + ai * ai
          write (f_temp, 902) xiA, sq / rnorm
      enddo
      close (f_temp)

      return
      end

C *************************************************************************
      subroutine psidump (ipot, zpsi)
C *************************************************************************
C Save wavepacket in raw form, for reading in by psiread().

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      integer f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /files/ f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3

C Local declarations:

      dimension zpsi(npts)

      open (f_temp, file='psi' // char(64 + ipot) // '.out')
      do 100 ix = 1, nXpts
        write (f_temp, *) zpsi(ix)
100   continue
      close (f_temp)

      return
      end

C *************************************************************************
      subroutine psiread (ipot, zpsi)
C *************************************************************************
C Read in wavepacket in raw form saved by psidump().

      implicit real*8 (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (npts = 2048)
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      integer f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3
      common /files/ f_in, f_out, f_normA, f_normB, f_matrix, f_temp,
     &  f_temp2, f_temp3

C Local declarations:

      dimension zpsi(npts)

      open (f_temp, file='psi' // char(64 + ipot) // '.inp')
      do 100 ix = 1, nXpts
        read (f_temp, *) zpsi(ix)
100   continue
      close(f_temp)

      return
      end

C *************************************************************************
      SUBROUTINE ANIONWF(ILEVEL,ZPSI)
C *************************************************************************
C      Anionwf stolen from READFCF8
C     Read the wavefuction off the fort.4 file of a
C      FCF program job (code of Ellison et al.)
C      This can actually pull off any wavefunction (excited vibrational
C      states) from the wavefunction calculated for the LOWER potential
C      using that program.  To use this feature change ILEVEL from 1 in
C      calling routine

C     This routine is mandatory if the user uses a general potential
C     from a file for the LOWER potential in the wavepacket calculation
C
       IMPLICIT real*8 (A-H,O-Y)
       Implicit Complex*16 (z)
       parameter (npts = 2048)
       DIMENSION NEN(2),PCOEFS(2,6),XKOUT(75)
       DIMENSION V(75,2),NPOT(2),E(2,50),VJ(2,70,50)
       DIMENSION ZPSI(npts)
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB

C
C Read data from file fort.4
C Lets allow the fort.4 file to have data about upper surface for compatibility
C
       OPEN(4,FILE='fort.4')
       READ(4,900) NOSC
  900  FORMAT(I3)
       IF(NOSC.EQ.1) THEN
        READ (4,903) NEN(1),N
       ELSE
        READ(4,905) NEN(1),NEN(2),N
       ENDIF
  903  FORMAT(2I3)
  905  FORMAT(3I3)
       NKNOT = N + 4
       READ(4,910) (XKOUT(I),I=1,NKNOT)
  910  FORMAT(13(6F12.6,/))
       READ(4,915) NPOT(1),(PCOEFS(1,I),I=1,6)
  915  FORMAT(I2,6F12.6)
       READ(4,920) (V(I,1),I=1,NKNOT)
  920  FORMAT((6F12.7))
```

```
      READ(4,925) (E(1,J),J=1,NEN(1))
  925 FORMAT(20F12.7)
      IF(NOSC.EQ.1) GO TO 10
      READ(4,915) NPOT(2),(PCOEFS(2,I),I=1,6)
      READ(4,920) (V(I,2),I=1,NKNOT)
      READ(4,925) (E(2,J),J=1,NEN(2))
   10 DO 20 I=1,NEN(1)
      READ(4,920) (VJ(1,J,I),J=1,N)
   20 CONTINUE
      IF(NOSC.EQ.1) GO TO 40
      DO 30 I=1,NEN(2)
      READ(4,920) (VJ(2,J,I),J=1,N)
   30 CONTINUE
      CLOSE(4)
      XH = XKOUT(2) - XKOUT(1)
      if ((xmin*0.529177 - xkout(1)) .gt. 0.0005) then
       write(6,*)'xmin= ',0.529177*dx
       write(6,*)'first knot at ',xkout(1)
       write(6,*)'first points dont match - Stopping'
       stop
      endif
      if ((xh-0.529177*dx) .gt.  0.0005) then
       write(6,*)'dx= ',0.529177*dx
       write(6,*)'knot spacing= ',xh
       write(6,*)'Grid sizes dont match - Stopping'
       stop
      endif
      if (nknot .ne. nxpts) then
      write(6,*)'nXpts= ',nxpts
      write(6,*)'nknots= ',nknots
      write(6,*)'Dont match - Stopping'
      endif
   40 write(*, *) ' DONE READING'

C This has stored all needed and uneeded data ......
C
C    Calculate wavefunction from spline coefficients
C     Want the ground state wavefunction, I=1:
      I=ILEVEL
      ZPSI(1)=VJ(1,1,I) + VJ(1,2,I)/4
      ZPSI(N)=VJ(1,N,I) + VJ(1,N-1,I)/4
      DO 100 J=2,N-1
        ZPSI(J)=VJ(1,J-1,I)/4 + VJ(1,J,I) + VJ(1,J+1,I)/4
  100 CONTINUE
C
C    Get the correct sign for wavefunction (+ at beginning)
C
      J=0
  105 J=J+1
      IF(real(ZPSI(J)).GT.0) GO TO 110
      IF(real(ZPSI(J)).EQ.0) GO TO 105
      DO 106 J=1,N
      VJ(1,J,I)=-VJ(1,J,I)
  106 ZPSI(J)=-ZPSI(J)
  110 CONTINUE

       return
       end

c *******************************************************
         subroutine morsewf(zpsi)
c *******************************************************
C MTZ's code for nlevel > 0 vibrational wavefunctions on a Morse potential
C (nlevel supplied in common block).

C Calculate the ground Wavefn for anharmonic oscillator
C Wavefn has following form ( see J. Res. N.B.S. A 65, 451 (1961))
C psi(x) = norm * [K * expval(x)]^(0.5*(K-1)) * exp(-0.5*K * expval(x))
C where :  expval(x) = exp(-beta*x)
C          norm   = sqrt ( beta/gamma(k-1) )

C This routine will only calculate lowest wavefunction of a
C Morse potential, for higher states (i.e. v=1 !) see the 2d code

C This method of calculating the wavefunctions of a Morse oscillator
C fails when the anharmonicity is very small (i.e. in the limit
C of a harmonic oscillator) because the gamma function blows up.
C This limit is reached for mildly anharmonic oscillators
C e.g. NCO- where we=2149 cm-1 and wexe=12.5 cm-1.
C In this case it is only a small approximation to use the H.O.
C wavefunction for the anion ground vibrational wavefunction

C Need to change two things to be able to calculate wavefunctions
C with small anharmonicities. 1) Gamma blows us, so get rid of it.
C By trial and error, Anorm = 10d-250 keeps the function in bounds.
C 2) (AK*expval)**((AK-1)/2) goes out of range.  Split it in two
C and multiply one half by exp(-AK*expval/2) first to keep it in bounds.

      implicit real*8 (a-h,o-y)
      implicit complex*16 (z)
      parameter (npts = 2048)
      character*80 wavefile
      dimension zpsi(npts),arg1(100),arg2(100),aLeg(0:100)
      dimension psinorm(100)
      common/const0/ zero,zeye, pi,c, twopi,sqrtpi,pisq,
     &  alessthanhalf
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB
      common /const2/ x0A, xomegA, v0A, xwexeA, dea, xalphaA, x0B,
     &  xomegB, xwexeB, deB, xalphaB, v0B, x0C, xomegC, xwexeC, deC,
     &  xalphaC, v0C, shelf, v0A_off, v0B_off, v0C_off
      common /const8/ hv1, fwhm1, dmu1, domega1, sech1, itype1, ipropA,
     &  ipropB, ipropC, iCpot, nlevel, Aoffset, Eint, fwhm2, thresh,
     &  proberange, ireadA, ireadB, irevB

C Form K = we/wexe and calculate gamma function of (k-1)

      AK = xomegA / xwexeA

c         open(2332,file='fuck.out')
c         open(2333,file='norm.out')

C evaluate the gamma fn.

          arg=AK-1.0
c         APOLY=1+1/(12.0*arg)+1/(288*arg*arg) - 139/(51840*arg**3)
c         gak=dsqrt(twopi/arg)*(arg**arg)*dexp(-arg)*APOLY
c         an = 10.0
          Anorm = 10d-250
c         Anorm=dsqrt(xalphaA/gak)
c       write(6,*)'Anorm, gak  = ',Anorm,gak
c         write(6,*)'Aleg(-1) = ',aLeg(0)

          do i=1,nxpts
             r=xmin+(dble(i)-1.0d0)*dx
           x=r-x0A
```

```
            expval=dexp(-xalphaA*x)
            splitexp = (AK*expval)**((AK-1)/4)*dexp(-AK*expval/2)
           zpsi(i)=dcmplx(Anorm * (AK*expval)**((AK-1)/4)*splitexp,0.0)
c            zpsi(i)=cmplx(Anorm * (AK*expval)**((AK-1)/2)*exp(-AK*expval/2),0.0)

            aLeg(0) = 1.0
            do iv = 1,nlevel
              arg3 = 1.0
              arg4 = 0.0
              do j = 0,iv-1
                arg3 = fact(iv-1)/(fact(j)*fact(iv-1-j))*fact(2*j)/fact(j+1)
                arg4 = arg4 + arg3 * aLeg(iv-j-1)
              enddo
              aLeg(iv) = AK*expval*aLeg(iv-1)-(AK-2*iv)*arg4
            enddo
            zpsi(i) = (1/(AK*expval)**nlevel)*aLeg(nlevel)*zpsi(i)
c           write(2332,*) r*0.529,real(zpsi(i))

899       format(e22.16,2x,e22.16)
          enddo

          call chknrm(zpsi,rnorm)
          do 11 i = 1,nXpts
            r = xmin+(i-1)*dx
            zpsi(i)=zpsi(i)/(rnorm)**0.5
c           write(2333,*) r*0.529,real(zpsi(i))
11      continue

c         close(2332)
c         close(2333)
          return
          end

C ***************************************************************************
        real*8 function fact(n)
C ***************************************************************************
C Calculates factorial of n.

          integer i

          fact = 1
          do i = 1,n
            fact = dble(i*fact)
          enddo
        return
          end

C ***************************************************************************
      subroutine make_abs
C ***************************************************************************
C Generate absorbing boundary functions for A, B surfaces

      implicit real*8 (a-h,o-y)
      implicit complex*16 (z)
      parameter (npts = 2048)
      common /abss/ absA(npts), absB(npts)
      common /const3/ tstep
      common/const4/xmin,xmax,nXpts,dx,npacket,iabsA,nxabsA,absfacA,
     &  iabsB,nxabsB,absfacB

      if (iabsA .eq. 1) then
        if (absfacA .le. 0) absfacA = (tstep / 1.3) * 0.0005
        do i = 1, nxabsA
          absA(i) = dexp(-absfacA * i * i)
        enddo
      endif

      if (iabsB .eq. 1) then
        if (absfacB .le. 0) absfacB = (tstep / 1.3) * 0.0005
        do i = 1, nxabsB
          absB(i) = dexp(-absfacB * i * i)
        enddo
      endif

      return
      end
```

8.2. trans2.2.1.f

```
C TRANS - Fourier transform program to obtain photoelectron spectra from
C dynamics matrix generated with ALPINE.

C Author:      B. Jeff Greenblatt
C Start date: March 1996 (for most recent edit, see last_date variable)
C Address:     Neumark Research Group
C              Department of Chemistry
C              University of California
C              Berkeley, CA 94720 USA
C Phone:       510-642-7761 or 510-643-9301
C E-mail:      jeff@radon.cchem.berkeley.edu

C INPUTS:  trans.inp     Input deck.
C          matrix.out    Overlap matrix (output from ALPINE).
C OUTPUTS: pwX.out       Photoelectron intensity vs. energy (X = 001, 002, ...
C                          -- one for each delay time).
C          trans.out     Copy of what's printed to screen during run.

C Notes for this version (add to bottom of list, please):

C Trans1.2: Uses correct power of two in array size of zmatrix (up to a maximum
C of nt2halfmax time elements). Also allows use of a position-dependent mu,
C though I am not at all sure if introducing mu(x) at this point in the simula-
C tion is theoretically legitimate.

C Trans1.3 (8dec96, BJG): Small change to check version header of matrix.out
C file being read, and also to read in an energy shift (eV) to apply to photon
C energy, to undo shifts applied in alpine propagation. Also changed name of
C input file to 'trans.inp'.

C 1.4 (11dec96) BJG: Added pwlist file to outputs, which lists letter with
C corresponding delay time.

C 1.5 (15dec96) BJG: Eliminated nwrap, nrange parameters -- instead program
C does the correction automatically. Also dropped mu() from program, eliminated
C some screen printing. Moved pwlist file to part of main output (dumps to
C screen.out). Updated headers -- now need two (for matrix.out and trans.inp).

C 1.6 (20dec96) BJG: Added convolution and regridding functions (specify
C directly in input deck). Changed labeling scheme of pwX.out files to numbers
C (pw001.out, etc.).

C 2.0 (16feb97) BJG: Switched from reading a wavefunction matrix to an overlap
C matrix, in anticipation of also being able to transform output from 2D
```

```
C propagation code ("TOBOGGAN"). 8jul97: Fixed a sporadic bug which seems to
C have had devastating consequences for certain potential surfaces: in calcu-
C lating e_min_pw, used nt2_max, not nt2half_max in expression. So ipts was 1
C point larger than it should have been -- thus overflowing memory. Result was
C that the first delay time spectrum was output correctly, but other were gar-
C bled. Also resulted in a slight (~few meV) shift in energy of output
C spectrum.

C 2.1 8jul97 BJG: Increased nt1max to 4096 to handle larger number of time
C delays.

C 2.1.1 BJG: Dump un-FFT'd spectrum to file in addition to normal photo-
C electron spectrum output.  File numbers = psiXYYY.ext, where YYY >
C 100.

C 2.2 8/21/97 BJG: To make compatible with aether (SGI system), changed
C text reading into integer reading for version numbers of input decks.
C This is better in the long run anyway.

C 2.2.1 1999-2-24 BJG: Corrected bug which made system crash if the num-
C ber of points in the final convoluted array is larger than nener (cur-
C rently 512). Now it catches error before kicking you out. Still uses
C version 2200 input deck.

C Global variable declarations:

      implicit double precision (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (nt1max = 4096, nt2max = 512, nt2halfmax = 256,
     &  nener = 512)
      dimension zmatrix(nt1max, nt2halfmax), et(nt1max),
     &  ztemp(nt2max), ztemp1(nt1max)
      common /pws/ pw(nener), pwconvol(nener)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq
      common /const1/ xmas, hb
      common /const4/ nt1, nt2, nt2half, nt2_fft, nt2half_fft,
     &  tmin, tstep
      common /const8/ hv2, fwhm2, sech2, e_shift, e_min, e_max,
     &    e_step, iconvol_type, e_ion, rmass, e_min_pw, e_max_pw,
     &    e_step_pw, ipts
      character*80 last_date
      common /version/ ivers_matrix, ivers_trans

C Set headers

      ivers_matrix = 2200
      ivers_trans = 2200
      last_date = '24 February 1999'

C Open output file which is carbon copy of what goes to the screen.

      open (2, file = 'trans.out')

C Greeting message.

900   format ('Welcome to TRANS: Fourier transform routine for ',
     &  'ALPINE outputs.')
901   format ('First written by B. Jeff Greenblatt, March 1996.')
910   format ('Input deck version = ', i5)
920   format ('Matrix.out version = ', i5)
930   format ('Last date edited = ', a)
      write (2, *)
      write (2, 900)
      write (2, 901)
      write (2, 910) ivers_trans
      write (2, 920) ivers_matrix
      write (2, 930) last_date
      write (6, *)
      write (6, 900)
      write (6, 901)
      write (6, 910) ivers_trans
      write (6, 920) ivers_matrix
      write (6, 930) last_date

C Read the needed data and also define some useful constants.

903   format ('Reading input deck.')
      write (2, 903)
      write (6, 903)
      call const()

C Read in overlap matrix.

904   format ('Reading matrix file.')
      write (2, 904)
      write (6, 904)
      call readzmatrix(zmatrix)

C Establish some parameters to use for workup later on.

      e_min_pw = hv2 - pi * (nt2half_fft - 1) / nt2half_fft
     &  / tstep
      e_max_pw = hv2 + pi / tstep
      e_step_pw = twopi / nt2_fft / tstep

C Generate automatic parameters, if needed.

      if ((e_min .eq. 0) .and. (e_max .eq. 0)) then
        e_min = e_min_pw
        e_max = e_max_pw
      endif
      if (e_step .eq. 0) then
        e_step = e_step_pw
      endif

      ipts = nint((e_max - e_min) / e_step) + 1 ! Final number of points

C Print some parameters.

100   format('Raw energy range and step size (eV) = ', f7.3, 2x,
     &  f7.3, 2x, f7.4)
      write (2, 100) e_min_pw * harev, e_max_pw * harev,
     &  e_step_pw * harev
      write (*, 100) e_min_pw * harev, e_max_pw * harev,
     &  e_step_pw * harev
110   format('Final energy range and step size (eV) = ', f7.3, 2x,
     &  f7.3, 2x, f7.4)
      write (2, 110) e_min * harev, e_max * harev,
     &  e_step * harev
      write (*, 110) e_min * harev, e_max * harev,
     &  e_step * harev
120   format('Number of points in raw and final arrays = ', i4,
     &  2x, i4)
      write(2, 120) nt2, ipts
      write(*, 120) nt2, ipts
```

```
      if (ipts .gt. nener) then
140     format('Number of final points exceeds ', i4, '. Change ',
     &     'nener in program.')
        write (*, 140) nener
        write (2, 140) nener
        stop
      endif

130   format('Convolution type = ', i1, '  Ion energy = ', f7.1,
     &  ' eV  Mass = ', f8.2, ' amu')
      write (2, 130) iconvol_type, e_ion * harev, rmass / amu
      write (*, 130) iconvol_type, e_ion * harev, rmass / amu

      write (2, *)
      write (*, *)

960   format('Table of delay times for spectrum files.')
902   format('File number  Delay (fs)')
      write(2, 960)
      write(2, 902)
      write(8, 960)
      write(*, 902)

C Main loop: read in delta t.

      do ideltat = 1, 99999
        read (1, *, end = 2) deltat
908     format (4x, i3, 6x, f10.2)
        write (2, 908) ideltat, deltat
        write (*, 908) ideltat, deltat
        deltat = deltat / atu

C Create E(t) vector to multiply by each column of zmatrix:

        t = tmin
        do it = 1, nt1
          et(it) = E2(t, deltat)
          t = t + tstep
        enddo

C Integrate O(t1, t2) * E(t1) * E(t1 - t2) over t1 for each t2:

        do it2 = 1, nt2half
          do it1 = 1, nt1
            if (it1 .lt. it2) then
              ztemp1(it1) = zero
            else
              ztemp1(it1) = zmatrix(it1, it2) * et(it1) *
     &          et(it1 - it2 + 1)
            endif
          enddo
          call zsimpint(nt1, ztemp1, tstep, z)
          ztemp(it2) = z
        enddo

C Fill in rest of array with 0's

        do it2 = nt2half + 1, nt2half_fft
          ztemp(it2) = zero
        enddo

C Now duplicate array on negative side, taking complex conjugate (this ensures
C FT is real):

        ztemp(nt2half_fft + 1) = zero
        do it2 = 2, nt2half_fft
          ztemp(nt2_fft - it2 + 2) = dconjg(ztemp(it2))
        enddo

C DEBUG: Save un-FFT'd spectrum to file:

        call openpw(3, ideltat+100)
        do i = 1, ipts
          re = dreal(ztemp(i))
          ai = dimag(ztemp(i))
          write (3, 906) (0 + (i - 1) * tstep) * atu,
     &      re
        enddo
        close (3)

C Perform fast Fourier transform, reordering points resulting from FFT since
C negative values are stored in second half of matrix.

        call FFT (ztemp, nt2_fft, 1)
        do it2 = nt2half_fft + 1, nt2_fft
          pw(it2 - nt2half_fft) = dreal(ztemp(it2))
        enddo
        do it2 = 1, nt2half_fft
          pw(it2 + nt2half_fft) = dreal(ztemp(it2))
        enddo

C Convolute with instrument response function.

        call convol()

C Write resulting spectrum to file.

        call openpw(3, ideltat)
        do i = 1, ipts
906       format (f14.6, 2x, f14.6)
          write (3, 906) (e_min + (i - 1) * e_step) * harev,
     &      pwconvol(i)
        enddo
        close (3)

C End loop.

      enddo

C Done.

950   format ('Done.')
2     write (2, 950)
      write (*, 950)

      close (1)
      close (2)

      stop
      end

C *****************************************************************************
      subroutine const ()
C *****************************************************************************
C Read the needed data and also define some useful constants. Order of
```

```
C input deck:

C   ivers:                          Version number of input deck. Must
C                                   match, or exits.
C   hv2, fwhm2:                     Photon energy (eV) and FWHM pulse
C                                   width (fs) of probe pulse.
C   iconvol_type, e_ion, rmass:     Convolution type (see convol() fn),
C                                   and parameters (eV and amu, respec-
C                                   tively) used by convolution routine.
C   e_min, e_max, e_step:           Range and step size (eV) desired for
C                                   convoluted spectra. e_step = 0 signals
C                                   to use the same step size as produced
C                                   by the Fourier transform; e_min =
C                                   e_max = 0 likewise signals to use
C                                   range of Fourier transform (useful if
C                                   not adding spectra from different
C                                   states together).
C   deltat:                         Pump-probe delay (fs). As many lines
C                                   of these can be entered as desired.

      implicit double precision (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (nt1max = 4096, nt2max = 512, nt2halfmax = 256,
     &  nener = 512)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq
      common /const1/ xmas, hb
      common /const4/ nt1, nt2, nt2half, nt2_fft, nt2half_fft,
     &  tmin, tstep
      common /const8/ hv2, fwhm2, sech2, e_shift, e_min, e_max,
     &   e_step, iconvol_type, e_ion, rmass, e_min_pw, e_max_pw,
     &   e_step_pw, ipts
      common /version/ ivers_matrix, ivers_trans

C Set conversion factors.

      harev = 27.211608d0
      evwn = 8065.479d0
      a0 = 0.52917706d0
      amu = 1822.882d0
      emu = 9.109534d-31
      harwn = harev * evwn
      amass = 1.66056d-27
      atu = 0.024199d0

C Set const0.

      zero = dcmplx (0.0d00, 0.0d00)
      zeye = dcmplx (0.0d00, 1.0d00)
      pi = dacos (-1.0d00)
      twopi = 2 * pi
      sqrtpi = dsqrt (pi)
      pisq = pi * pi

C Speed of light in cm/s and hbar in atomic units.

      c = 2.99792458d10
      hb = 1.0d0

C Sechfactor is multiplied by time inside sech^2 so that relative pulse intensi-
C   ty = 0.5 when time = FWHM / 2.

      sechfactor = 1.762747174d00
```

```
C Begin reading input deck.

      open (1, file='trans.inp')

C Read version header

      read (1, *) ivers
      if (ivers .ne. ivers_trans) then
        close(1)
903     format('Version does not match: ', i5, ' sought, ', i5,
     &    ' read. Aborting.')
        write(2, 903) ivers_trans, ivers
        write(*, 903) ivers_trans, ivers
        stop
      endif

      read (1, *) hv2, fwhm2
      hv2 = hv2 / harev
      fwhm2 = fwhm2 / atu
      sech2 = sechfactor / fwhm2

C Read convolution parameters.

      read (1, *) iconvol_type, e_ion, rmass
      e_ion = e_ion / harev
      rmass = rmass * amu

C Read energy rebinning parameters.

      read (1, *) e_min, e_max, e_step
      e_min = e_min / harev
      e_max = e_max / harev
      e_step = e_step / harev

      return
      end

C *************************************************************************
      subroutine readzmatrix(zmatrix)
C *************************************************************************
C Read overlap matrix into zmatrix. Format of file:

C Version: a string indicating which version of alpine produced file.

C nqueue: number of points in shorter time dimension (overlap length).

C tmin, tstep: initial time (fs) and time step (fs) between each wavefunction.

C e_shift: amount (eV) to shift photon energy in input deck, due to shifts
C   applied to potentials in alpine propagation for more efficient math.

C Remaining lines: complex format data of matrix.

      implicit double precision (A-H, O-Y)
      implicit complex*16 (Z)
      parameter (nt1max = 4096, nt2halfmax = 256)
      dimension zmatrix(nt1max, nt2halfmax)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq
      common /const1/ xmas, hb
      common /const4/ nt1, nt2, nt2half, nt2_fft, nt2half_fft,
     &  tmin, tstep
```

```
      common /const8/ hv2, fwhm2, sech2, e_shift, e_min, e_max,
     &    e_step, iconvol_type, e_ion, rmass, e_min_pw, e_max_pw,
     &    e_step_pw, ipts
      common /version/ ivers_matrix, ivers_trans

C Open file and read header information.

      open (3, file='matrix.out')

      read (3, *) ivers
      if (ivers .ne. ivers_matrix) then
903     format('Version does not match: ', i5, ' sought, ', i5,
     &     ' read. Aborting.')
        write(2, 903) ivers_matrix, ivers
        write(6, 903) ivers_matrix, ivers
        close(3)
        stop
      endif

      read (3, *) nt2half
905   format('Half nt2 = ', i4)
      write(2, 905) nt2half
      write(*, 905) nt2half
      if (nt2half .gt. nt2halfmax) then
902     format('Half nt2 exceeds allotted memory (Increase nt2halfmax ',
     &     'in source code). Terminating...')
        write(2, 902)
        write(6, 902)
        close(3)
        stop
      endif

      read (3, *) tmin, tstep
      tmin = tmin / atu
      tstep = tstep / atu

      read (3, *) e_shift
      e_shift = e_shift / harev

C Apply shift to photon energy.

      hv2 = hv2 + e_shift

C Read data.

      do it1 = 1, nt1max + 1 ! Loop normally broken at read statement.
        if (it1 .gt. nt1max) then
901       format('Nt1 exceeds allotted memory (Increase nt1max ',
     &       'in source code). Terminating...')
          write(2, 901)
          write(6, 901)
          stop
        endif

C Read in matrix elements, creating diagonal for first nt2 columns:
        it2max = it1
        if (it2max .gt. nt2half) then
          it2max = nt2half
        endif
        do it2 = 1, it2max
          read (3, *, end = 1) zmatrix(it1, it2)
        enddo
        do it2 = it2max + 1, nt2half
          zmatrix(it1, it2) = zero
        enddo
      enddo

C Calculate correct number of wavefunctions read.

1     close (3)
      nt1 = it1 - 1

C Print total wavefunctions read.

900   format('Nt1 = ', i4)
      write(2, 900) nt1
      write(*, 900) nt1

C Find appropriate power of two size for fft array.

      nt2half_fft = nt2halfmax
      do while (nt2half .le. nt2half_fft / 2)
        nt2half_fft = nt2half_fft / 2
      enddo

      nt2_fft = 2 * nt2half_fft

      return
      end

C *********************************************************************
      subroutine FFT (x, n, isign)
C *********************************************************************
C
C *************************************************************
C * The fft computes the discrete fast Fourier transform of a   *
C * sequence of n terms.                                        *
C * The forward FFT computes                                    *
C *       y(j)= sum (from k=0 to n-1) x(k)*exp(2*pi*i*j*k/n)    *
C * the backward FFT computes                                   *
C *         y(j)= sum (from k=0 to n-1) x(k)*exp(-2*pi*i*j*k/n) *
C *                                                             *
C *  x is a complex array of length n.                          *
C *  n is a power of 2.  n<=16384                               *
C * isign is the direction of the transform.  If isign >= 0 then*
C * the fft is forward , otherwise backward.                    *
C *                                                             *
C *  Ref.  Cooley, Lewis, Welch.  The FFT and its applications  *
C *        IEEE Trans. on Education, vol. E-12 #1; p. 29        *
C *************************************************************

      implicit double precision (A-H, O-Y)

C Local declarations:

      complex*16  s, v, w, x(n), cstore(16384)
      data ntbl/0/

C The roots of unity exp(pi*i*k/j) for j=1,2,4,..,n/2 and k=0,1,2,..,j-1
C are computed once and stored in a table.
C This table is used in subsequent calls of fft with parameter n<=ntbl

      if (n .gt. ntbl) then
        ntbl = n
        pi = 3.14159265358979d00
        j = 1
```

```
        icnt = 0
10      s = pi * (0, 1) / j
        do 20 k = 0, j - 1
          icnt = icnt + 1
20      cstore(icnt) = exp(s * k)
        j = j + j
        if (j .lt. n) goto 10
      endif

C  ******Bit reversal**********
C The x(j) are permuted in such a way that each new place number j is
C the bit reverse of the original placenumber.

      j = 1
      do 30 i = 1, n
        if (i .le. j) then
          v = x(j)
          x(j) = x(i)
          x(i) = v
        endif
        m = n / 2
25      continue
        if (j .gt. m) then
          j = j - m
          m = m / 2
          if (m .ge. 1) go to 25
        else
          j = j + m
        endif
30    continue

C ************Matrix multiplication***************
C The roots of unity and the x(j) are multiplied.

      j = 1
      icnt = 0
40    jj = j + j
      do 50 k = 1, j
        icnt = icnt + 1
        w = cstore(icnt)
        if (isign .lt. 0) w = dconjg (w)
        do 50 i = k, n, jj
          v = w * x(i + j)
          x(i + j) = x(i) - v
50        x(i) = x(i) + v
      j = jj
      if (j .lt. n) goto 40

      return
      end

C *************************************************************************
      function E2(t, deltat)
C *************************************************************************
C Returns envelope function of E-field of laser pulse 2 at time t, given delay
C time deltat. Note that cosh term is not squared; I = E^2 = 1/cosh^2, but this
C is calculating just E = 1/cosh.

      implicit double precision (A-H, O-Y)
      implicit complex*16 (Z)
      common /const8/ hv2, fwhm2, sech2, e_shift, e_min, e_max,
     &   e_step, iconvol_type, e_ion, rmass, e_min_pw, e_max_pw,
     &   e_step_pw, ipts

      E2 = 1 / cosh (sech2 * (t - deltat))

      return
      end

C *************************************************************************
      subroutine openet(ifile, icount)
C *************************************************************************
C Opens et file.

      implicit double precision (A-H, O-Y)
      implicit complex*16 (Z)

      open (ifile, file='et' // char(64 + icount) // '.out')

      return
      end

C *************************************************************************
      subroutine openpw(ifile, icount)
C *************************************************************************
C Opens pw file using three-digit file code for icount.

      implicit double precision (A-H, O-Y)
      implicit complex*16 (Z)

      open(ifile, file='pw' // char(48 + icount / 100) // char(48
     &  + mod(icount / 10, 10)) // char(48 + mod(icount, 10)) //
     &  '.out')
c     open (ifile, file='pw' // char(64 + icount) // '.out')

      return
      end

C *************************************************************************
      subroutine convol()
C *************************************************************************
C BJG 19dec96: Revised from convol.f (operated on file). Convolutes raw FFT
C spectrum with an instrument response function; also rebins data to user's
C specifications (e_min, e_max, e_step). Several convolution types are avail-
C able (iconvol_type):

C 0 : no convolution.
C 1 : simple tophat distribution.
C 2 : isotropic distribution in magnetic bottle (see B. J. Greenblatt et al.,
C     Chem. Phys. Lett. 258, 523 (1996).
C 3 : sin^2 distribution (sideways peaked)
C 4 : cos^2 distribution (forward-backward peaked)

      implicit double precision (a-h,o-y)
      implicit complex*16 (Z)
      parameter (nener = 512)
      common /pws/ pw(nener), pwconvol(nener)
      dimension pwtemp(nener)
      common /convert/ harev, evwn, a0, amu, emu, harwn, amass, atu
      common /const4/ nt1, nt2, nt2half, nt2_fft, nt2half_fft,
     &  tmin, tstep
      common /const8/ hv2, fwhm2, sech2, e_shift, e_min, e_max,
     &   e_step, iconvol_type, e_ion, rmass, e_min_pw, e_max_pw,
     &   e_step_pw, ipts
      common /params/ ak, aeoffset
```

```
C First check that distribution type is not 0; simply interpolate array to
C new grid if so.

      if (iconvol_type .eq. 0) then
        call regrid()
        return
      endif

C Establish constants (in electron mass units)

      electronmass = 1

C Calculate constants for our distribution function

      ak = rmass / (4 * electronmass * e_ion)
      aeoffset = e_ion * electronmass / rmass

C Blank out target array.

      do j = 1, ipts
        pwconvol(j) = 0
      enddo

C Perform convolution. i indexes points in original array (pw); j indexes
C points in new array (pwconvol). Erange is range of convolution function,
C which changes with energy. anorm is accumulated weight which is then used to
C normalize each convoluted point.

      do i = 1, nt2_fft

        en = e_min_pw + (i - 1) * e_step_pw ! Energy of original array
        erange = dsqrt(en / ak)
        j_min = nint((en + aeoffset - erange - e_min)
     &    / e_step) + 1
        j_max = nint((en + aeoffset + erange - e_min)
     &    / e_step)

C First record distribution function in array and add up norm.

        anorm = 0
        do j = j_min, j_max
          dist_temp = dist(en, e_min + (j - 1) * e_step)
          if ((j .ge. 1) .and. (j .le. ipts)) then
            pwtemp(j) = dist_temp
          endif
          anorm = anorm + dist_temp
        enddo

C Now copy weighted and normalized distribution to final array.

        if (anorm .gt. 0) then
          a =  pw(i) * e_step_pw / e_step / anorm
          do j = j_min, j_max
            if ((j .ge. 1) .and. (j .le. ipts)) then
              pwconvol(j) = pwconvol(j) + pwtemp(j) * a
            endif
          enddo
        endif

      enddo

      return
      end
```

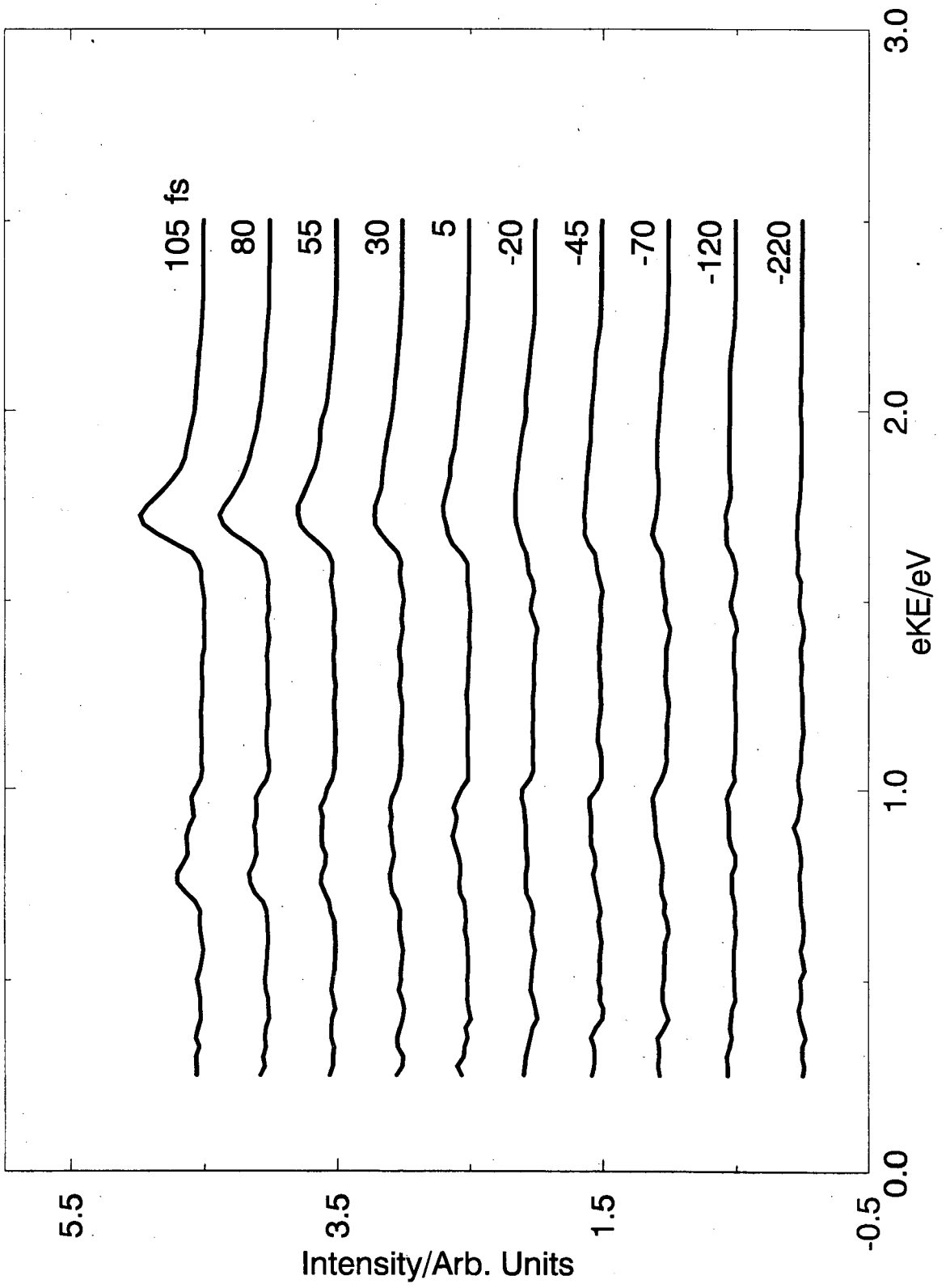
```
C ***********************************************************************
      function dist(ereal, eapp)
C ***********************************************************************
C Return distribution function value (ereal = real eKE, eapp = apparent eKE).
C Convol_type (in common block) specifies which distribution to use; for
C details, see comments under convol() subroutine. Normalization not required
C here, as convol() does its own normalization on each point.

      implicit double precision (a-h,o-y)
      implicit complex*16 (Z)
      common /const8/ hv2, fwhm2, sech2, e_shift, e_min, e_max,
     &   e_step, iconvol_type, e_ion, rmass, e_min_pw, e_max_pw,
     &   e_step_pw, ipts
      common /params/ ak, aeoffset

      if (iconvol_type .eq. 1) then ! Tophat
        dist = 1
      else
        temp = eapp - aeoffset - ereal
        r = ak * temp * temp / ereal
        if (r .lt. 1) then
          if (iconvol_type .eq. 2) then ! Isotropic
            dist = dsqrt(1 - r)
          else
            if (iconvol_type .eq. 3) then ! Sin^2
              dist = (1 - r) * dsqrt(1 - r)
            else
              if (iconvol_type .eq. 4) then ! Cos^2
                dist = r * dsqrt(1 - r)
              else
900             format ('iconvol_type not supported. ',
     &            'Terminating.')
                write (2, 900)
                write (*, 900)
                stop
              endif
            endif
          endif
        else ! Extra trap to ensure don't drop below zero in dsqrt.
          dist = 0
        endif
      endif

      return
      end


C ***********************************************************************
      subroutine regrid()
C ***********************************************************************
c Interpolates pw array, with grid parameters e_min_pw, e_max_pw, e_step_p,
C onto pwconvol array using new grid parameters e_min, e_max, e_step.

      implicit double precision (a-h,o-y)
      implicit complex*16 (Z)
      parameter (nt1max = 4096, nt2max = 512, nt2halfmax = 256,
     &  nener = 512)
      common /const4/ nt1, nt2, nt2half, nt2_fft, nt2half_fft,
     &  tmin, tstep
      common /const8/ hv2, fwhm2, sech2, e_shift, e_min, e_max,
     &   e_step, iconvol_type, e_ion, rmass, e_min_pw, e_max_pw,
     &   e_step_pw, ipts
```

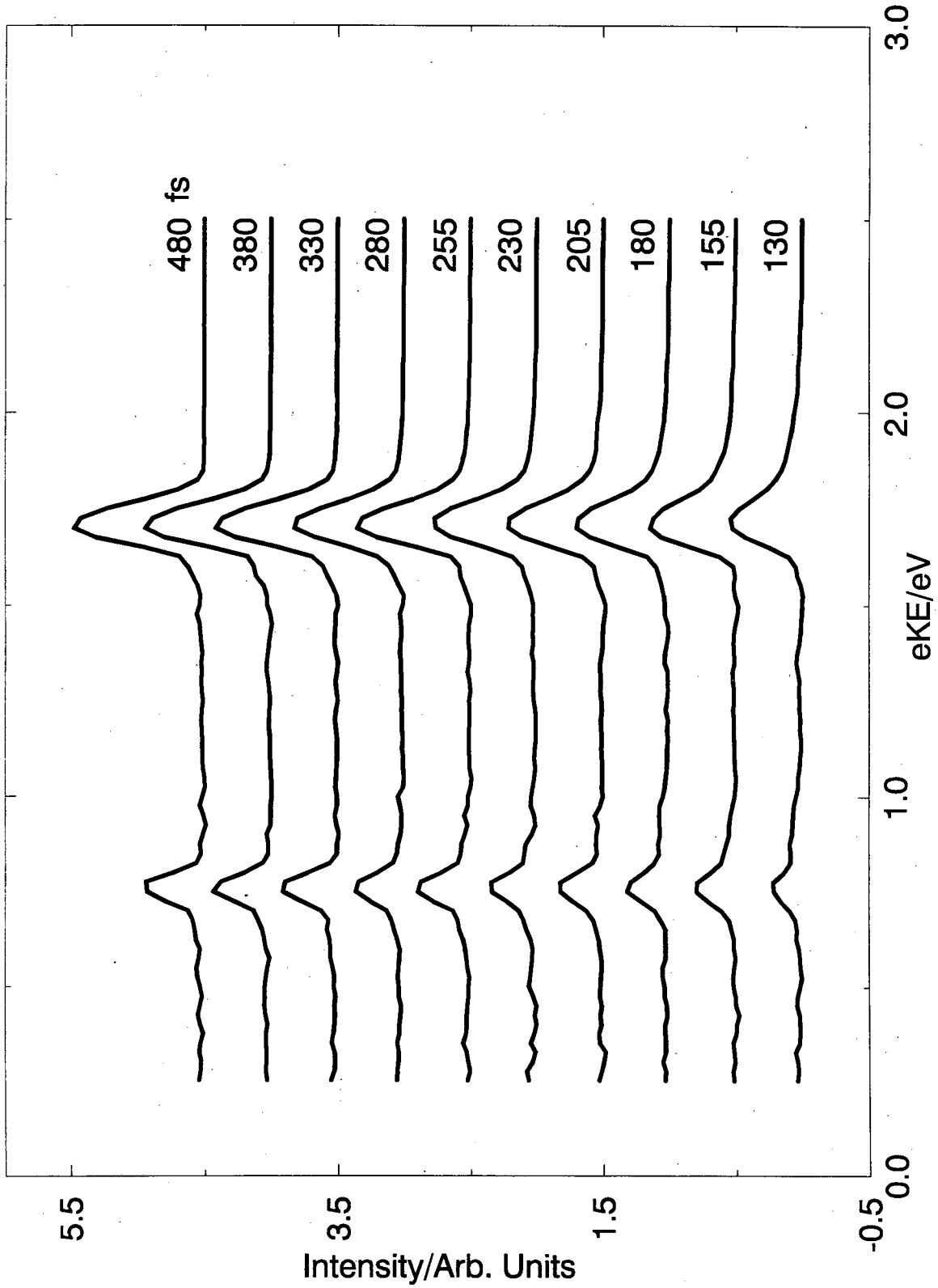
```
      common /pws/ pw(nener), pwconvol(nener)

      do i = 1, ipts
        en = e_min + (i - 1) * e_step ! Energy in new array
        j = int((en - e_min_pw) / e_step_pw) + 1 ! Lgest old array elem <= en
        if ((j .lt. 1) .or. (j .ge. nt2_fft)) then
          pwconvol(i) = 0
        else
          en_pw = e_min_pw + (j - 1) * e_step_pw ! Old array energy
          pwconvol(i) = (pw(j) * (en_pw + e_step_pw - en) +
     &      pw(j + 1) * (en - en_pw)) / e_step_pw
        endif
      enddo

      return
      end

C *************************************************************************
      subroutine zsimpint(nx, zf1, dx, zint)
C *************************************************************************
C Complex Simpson's rule integrator.

      implicit double precision (A-H, O-Y)
      implicit complex*16 (Z)

C Local declarations:

      parameter (nypts = 4096)
      dimension zf1(nypts), zf2(nypts)

C Define:

      dx1 = dx
      dx2 = dx * 2.
      ixn = 0.

      if (nx .gt. nypts) then
        write (6,*) ' zsimpint : nx .gt. nypts = ', nypts
      endif

      if ((mod(nx, 2) .eq. 0)) then
        nx1 = nx - 1
        nx2 = 0.50d00 * nx1 + 1
        zint = 0.50d00 * dx * (zf1(nx - 1) + zf1(nx))
      else
        nx1 = nx
        nx2 = 0.50d00 * nx1 + 1
        zint = 0.0d00
      endif

C Copy the odd elements of zf1 array into zf2.

      do 10 ix = 1, nx1, 2
        ixn = ixn + 1
10      zf2(ixn) = zf1(ix)

C Now integrate zf1, zf2 in two pieces.

      call ztrapint(nx1, zf1, dx1, zint1)
      call ztrapint(nx2, zf2, dx2, zint2)
      zint = zint + (4.0d00 * zint1 - zint2) / 3.0d00

      return
      end

C *************************************************************************
      subroutine ztrapint(npts, zf, dx, zint)
C *************************************************************************

      implicit double precision (A-H, O-Y)
      implicit complex*16 (Z)
      common /const0/ zero, zeye, pi, c, twopi, sqrtpi, pisq

C Local declarations:

      dimension zf(npts)

C Trapezoidal rule integrator for f(1) - f(npts) <-> f(x0) - f(xf).

      zint = zero

      do 100 i = 2, npts - 1
        zint = zint + zf(i)
100   continue
      zint = zint + (zf(1) + zf(npts)) / 2.0d00
      zint = zint * dx

      return
      end
```

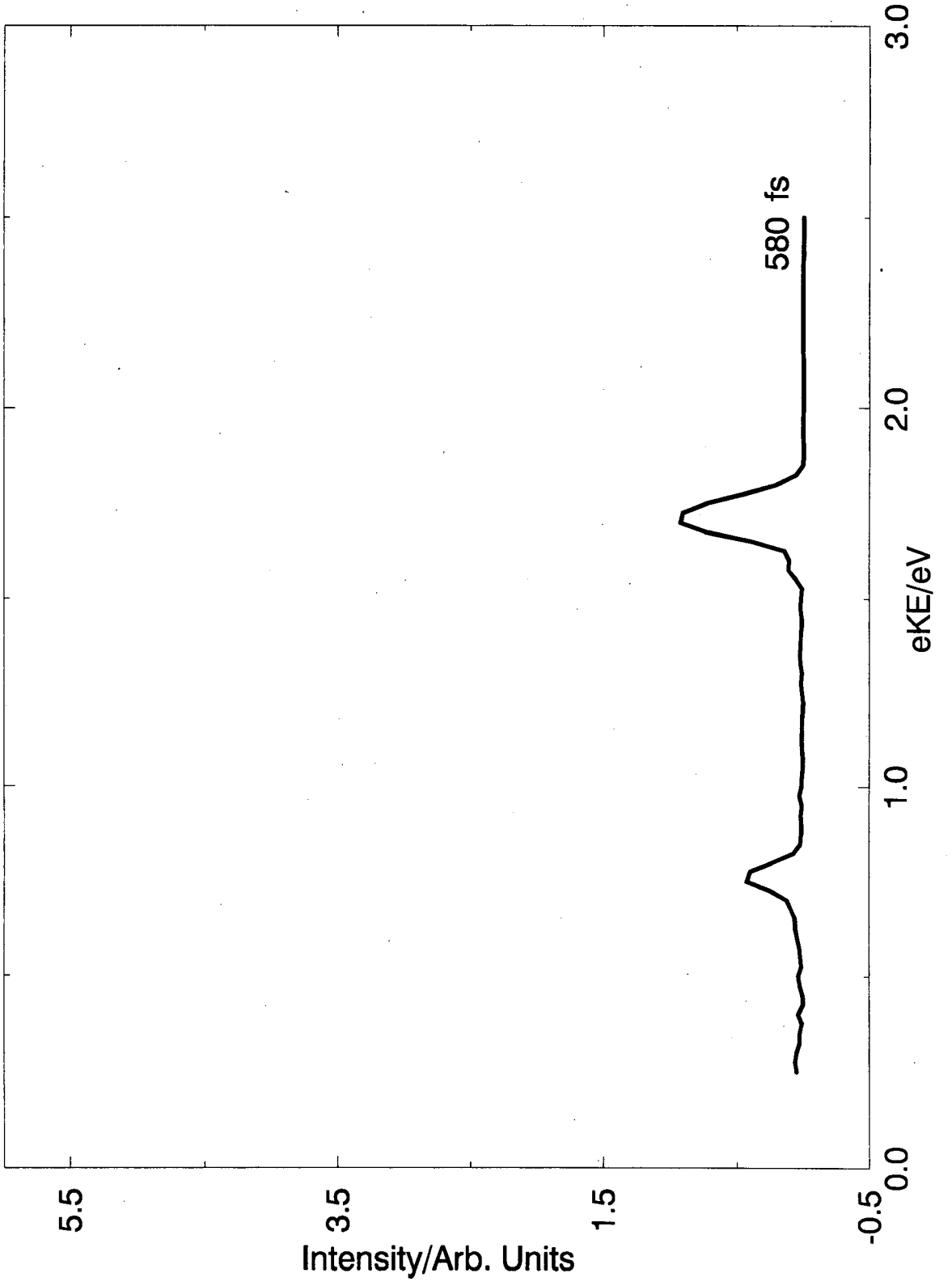
Appendix 3. Complete FPES spectra of $I_2^-(Ar)_n$ and $I_2^-(CO_2)_n$ clusters

Contained in this Appendix is the complete set of FPES spectra collected for Chapters 6 and 7. Chapters 4 [$I_2^-(Ar)_6$ and $I_2^-(Ar)_{20}$] and 5 [I_2^-, $I_2^-(CO_2)_4$ and $I_2^-(CO_2)_{16}$] use segments of the same data set, but the $I_2^-(CO_2)_n$ spectra have been modified to remove background signals more effectively, and the zero of time of most of the spectra has changed slightly. The bare I_2^- spectra appearing in Chapters 3 and 4 differ from the I_2^- spectra here, as they were acquired without pulsed deceleration. These I_2^- spectra are not included. The $I_2^-(Ar)_6$ spectra are presented as two sets. The first set had good signal-to-noise, but sparse representation beyond 1.2 ps. The second set, while denser at long time delays, was not suitable for presentation in papers due to poor background subtraction.

1. I_2^-

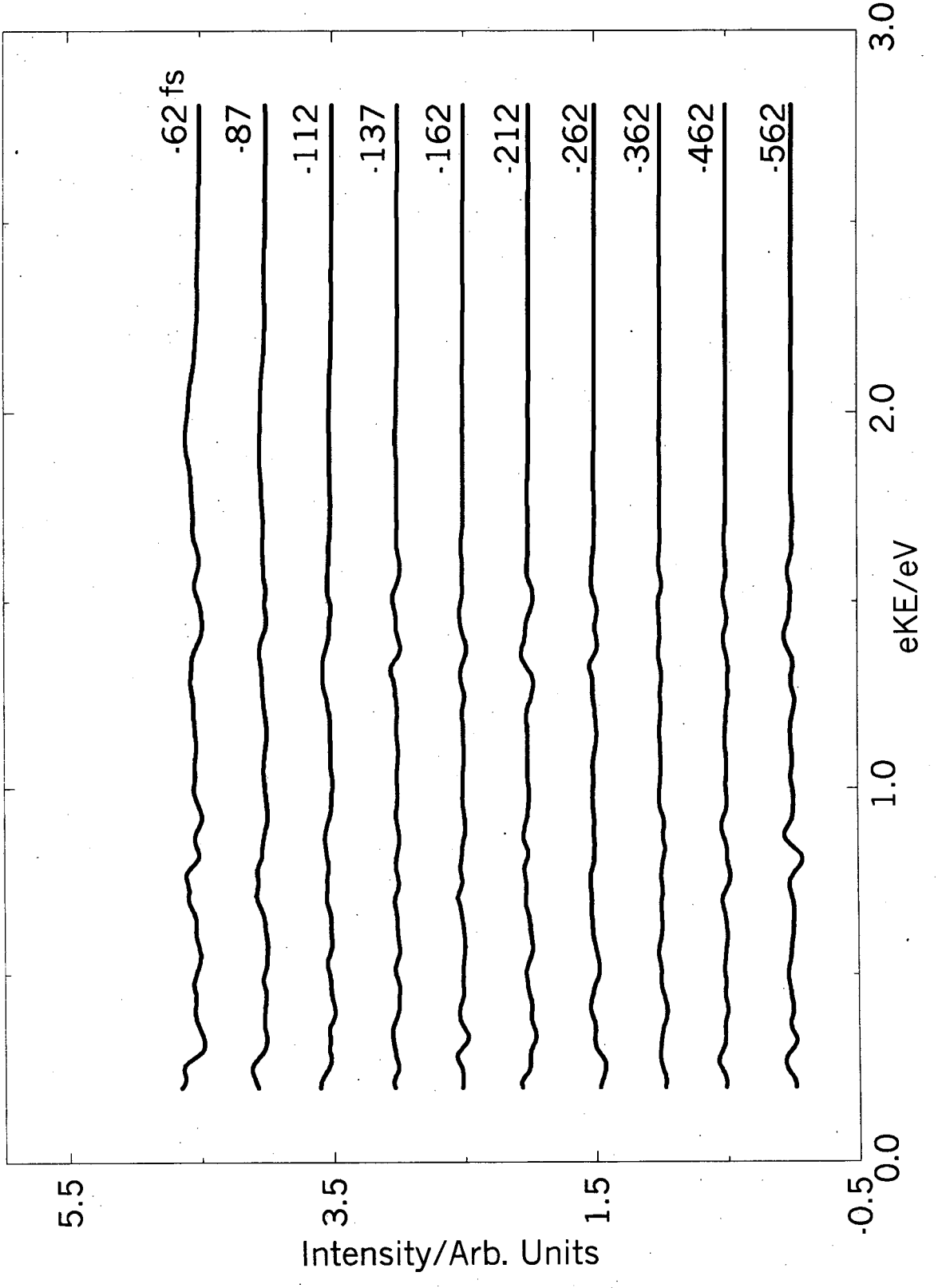

105 fs
80
55
30
5
-20
-45
-70
-120
-220
3.0
2.0
1.0
0.0
eKE/eV
5.5
3.5
1.5
-0.5
Intensity/Arb. Units

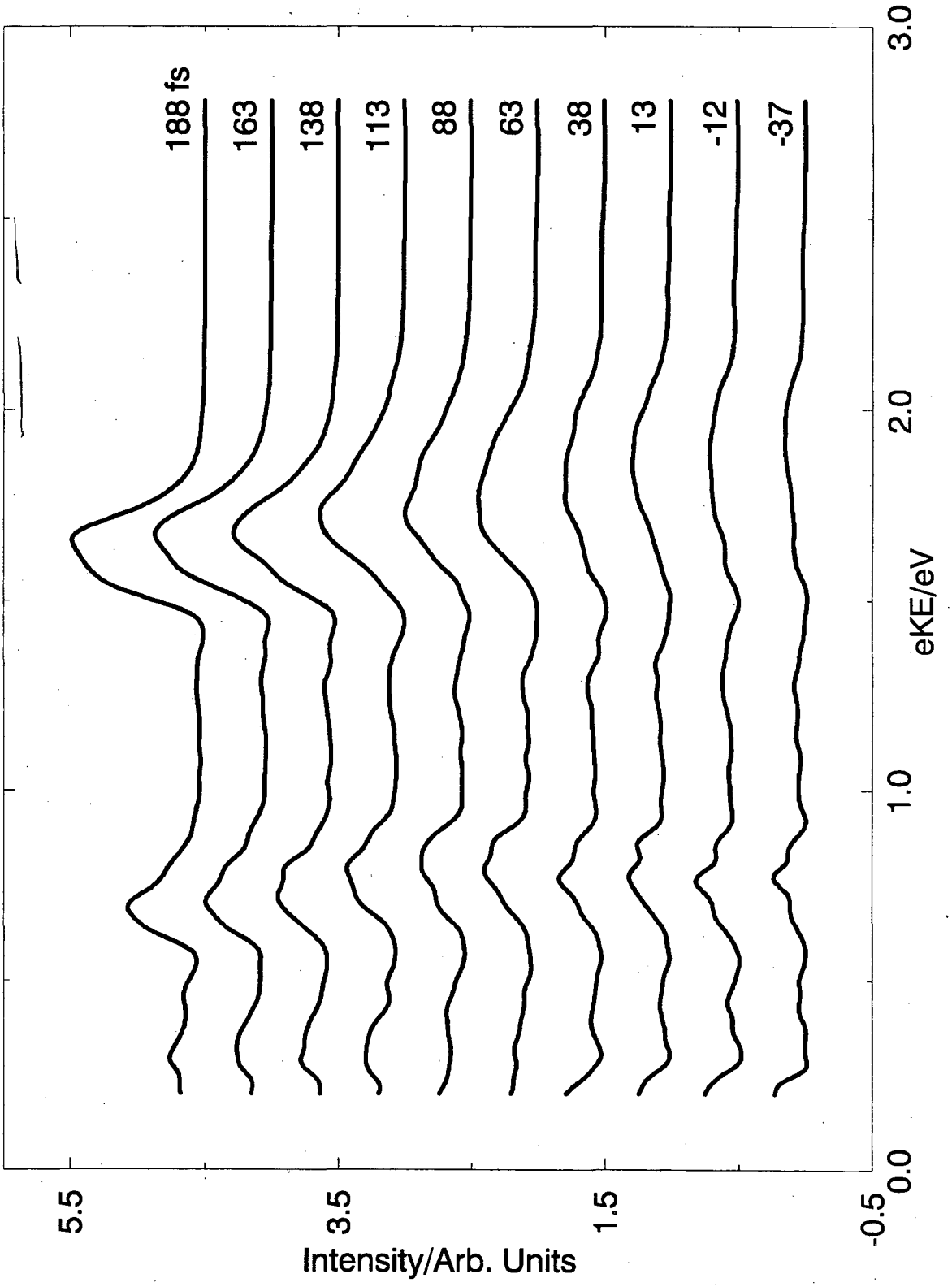

480 fs
380
330
280
255
230
205
180
155
130
5.5
3.5
1.5
-0.5
Intensity/Arb. Units
0.0
1.0
2.0
3.0
eKE/eV

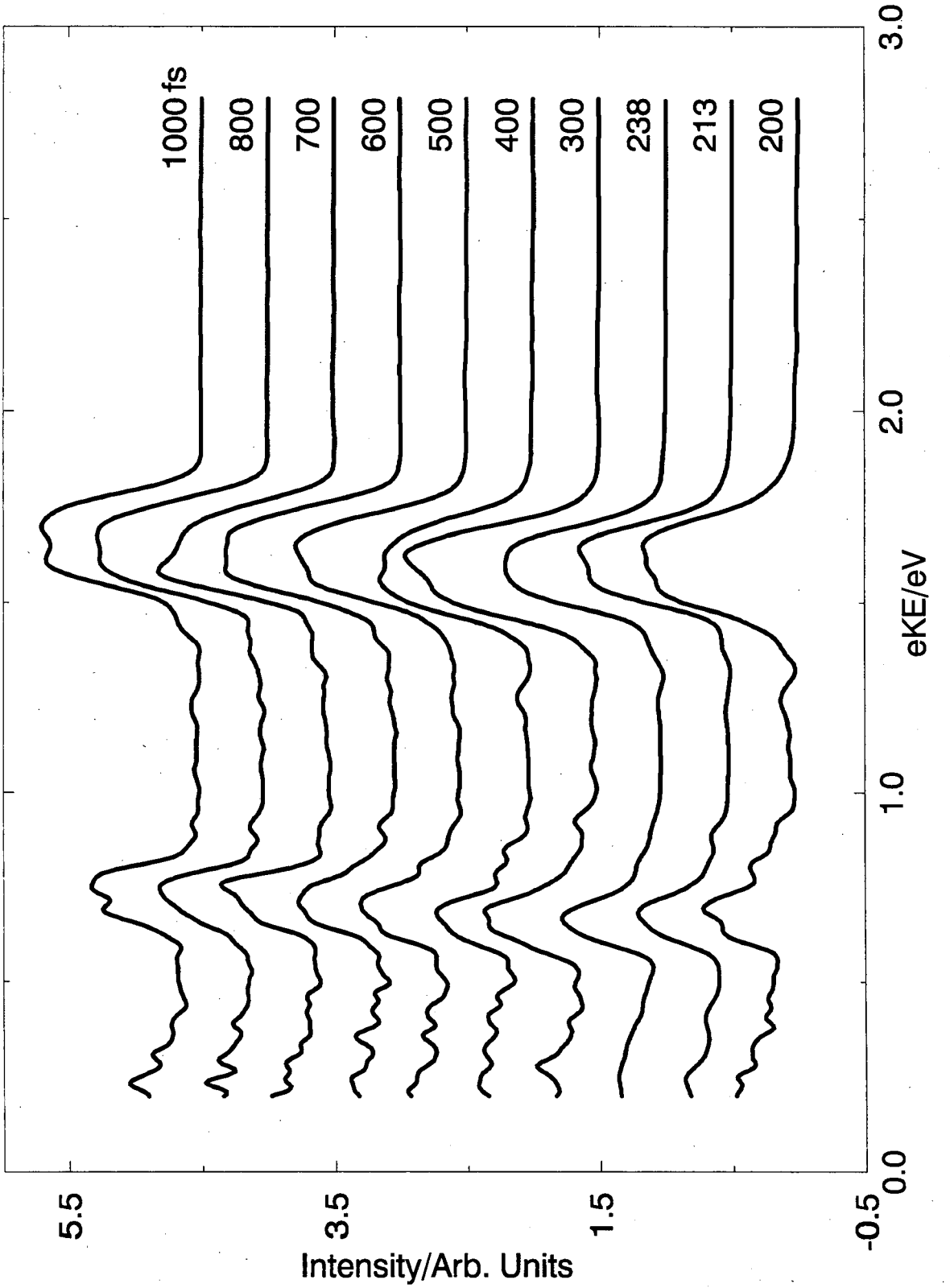

580 fs
eKE/eV
3.0
2.0
1.0
0.0
-0.5
5.5
3.5
1.5
Intensity/Arb. Units

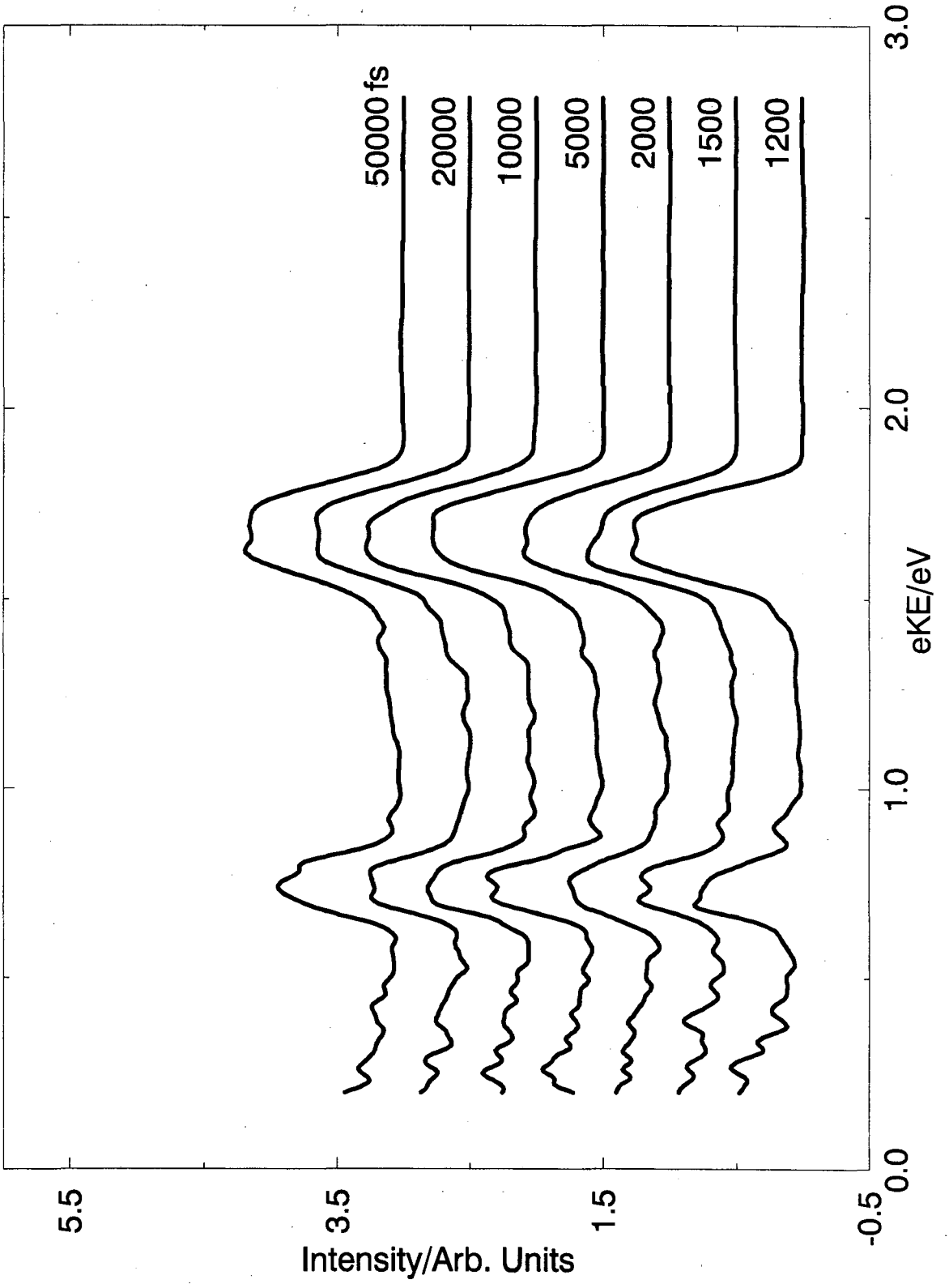
2. $I_2^-(Ar)_6$

First set

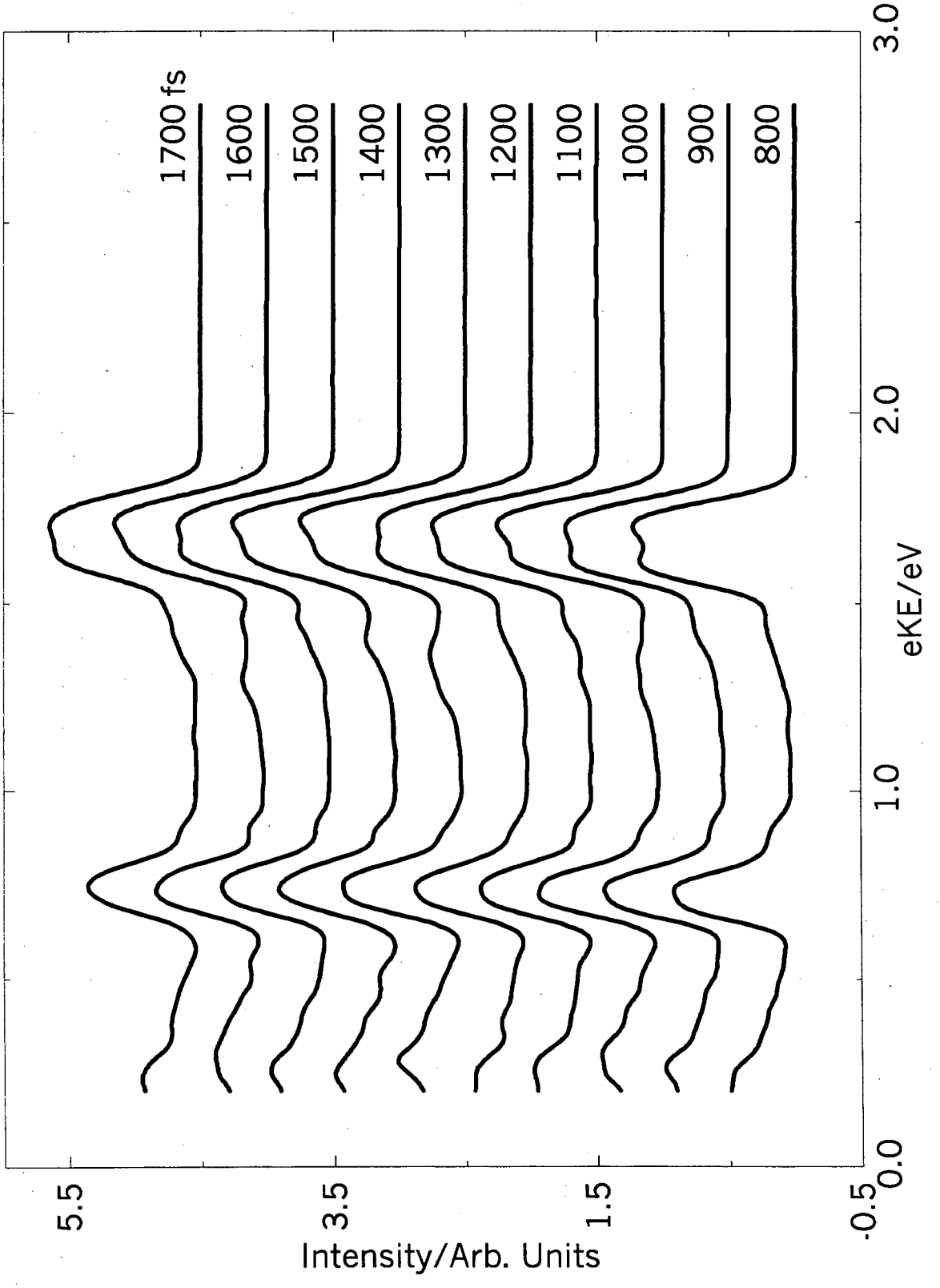


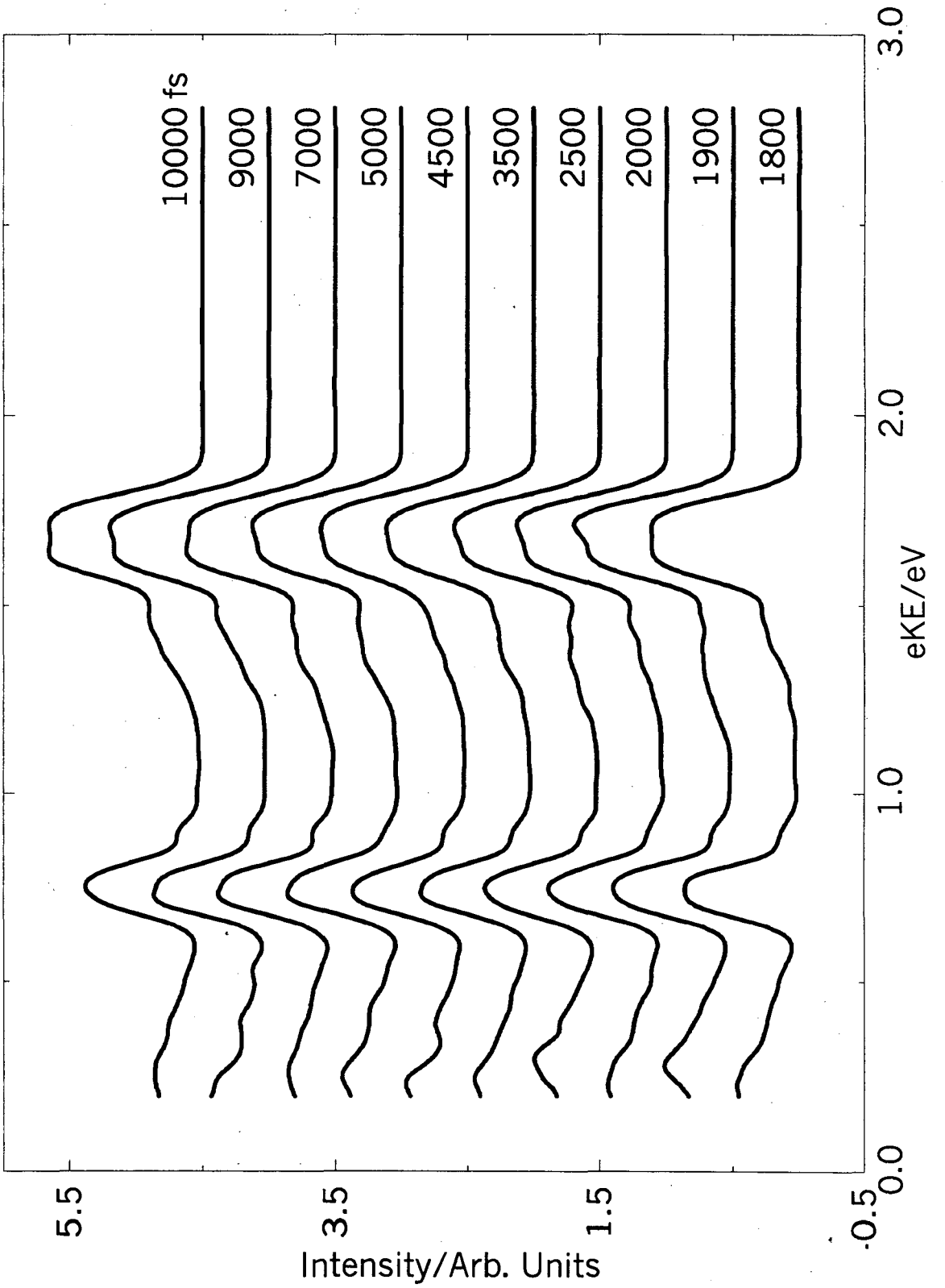

188 fs
163
138
113
88
63
38
13
-12
-37
3.0
2.0
1.0
0.0
eKE/eV
5.5
3.5
1.5
-0.5
Intensity/Arb. Units

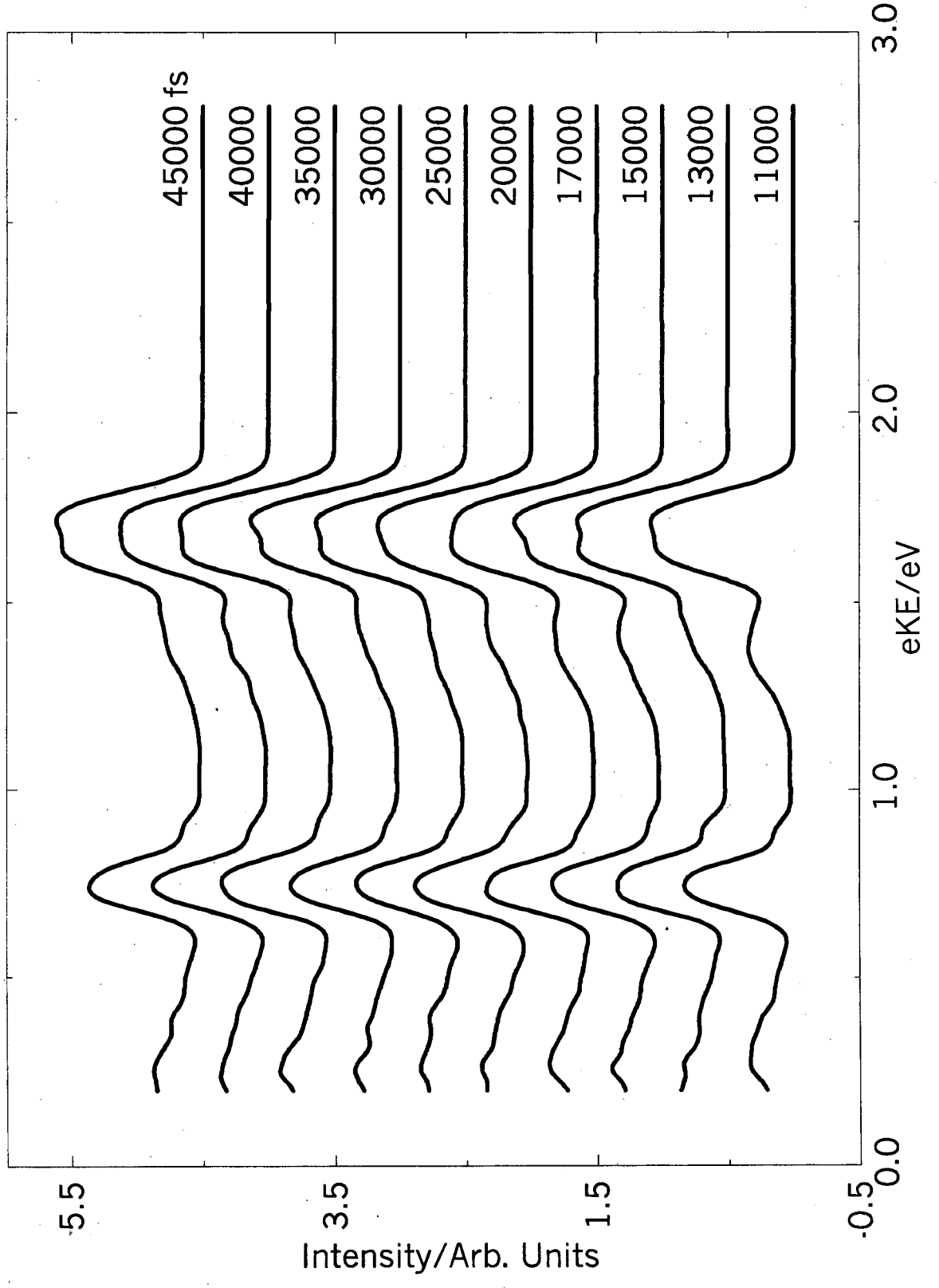

1000 fs
800
700
600
500
400
300
238
213
200
3.0
2.0
1.0
0.0
eKE/eV
5.5
3.5
1.5
-0.5
Intensity/Arb. Units

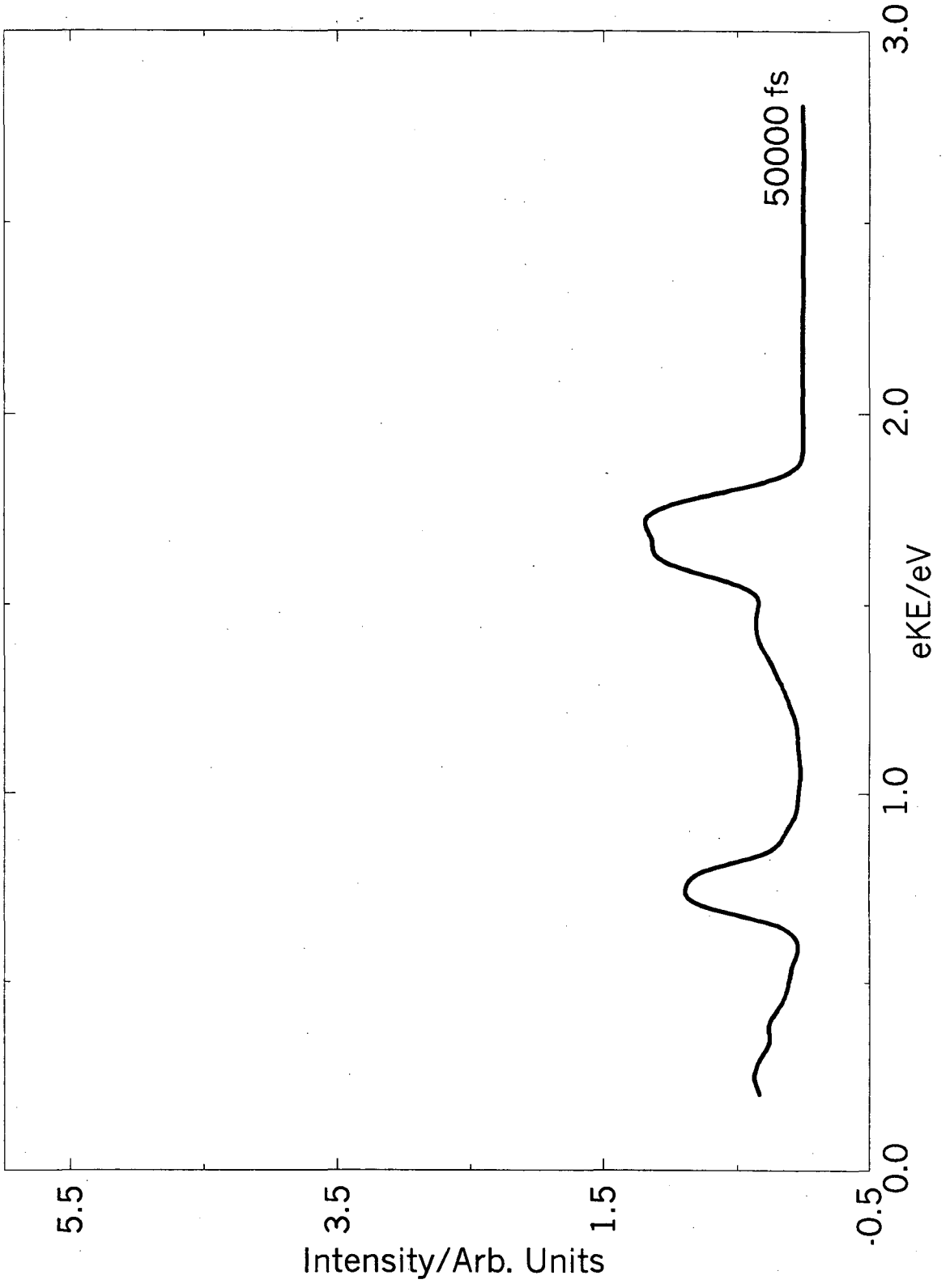

50000 fs
20000
10000
5000
2000
1500
1200
3.0
2.0
1.0
0.0
eKE/eV
5.5
3.5
1.5
-0.5
Intensity/Arb. Units

Second set

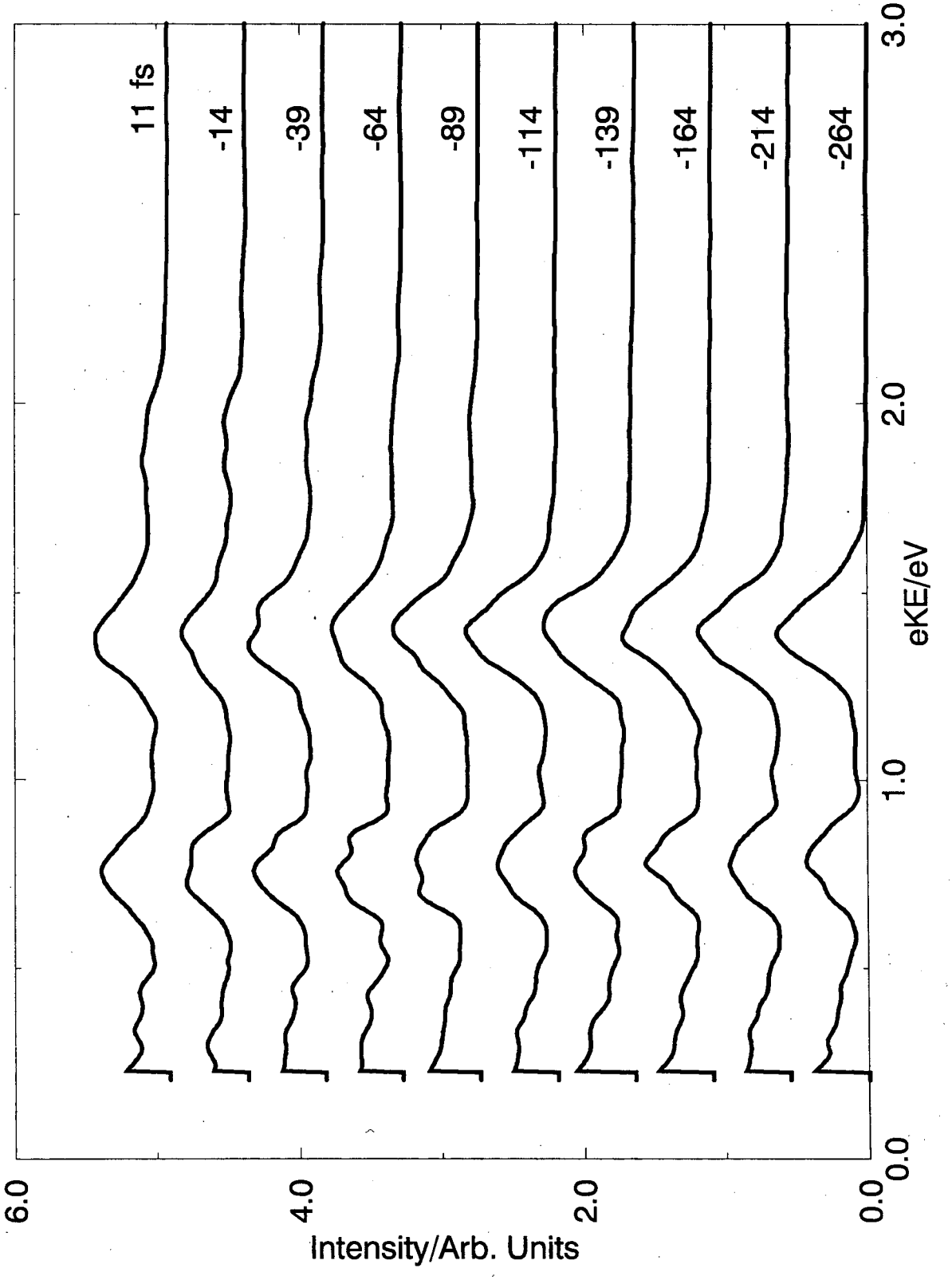

1700 fs
1600
1500
1400
1300
1200
1100
1000
900
800
3.0
2.0
1.0
0.0
eKE/eV
5.5
3.5
1.5
-0.5
Intensity/Arb. Units

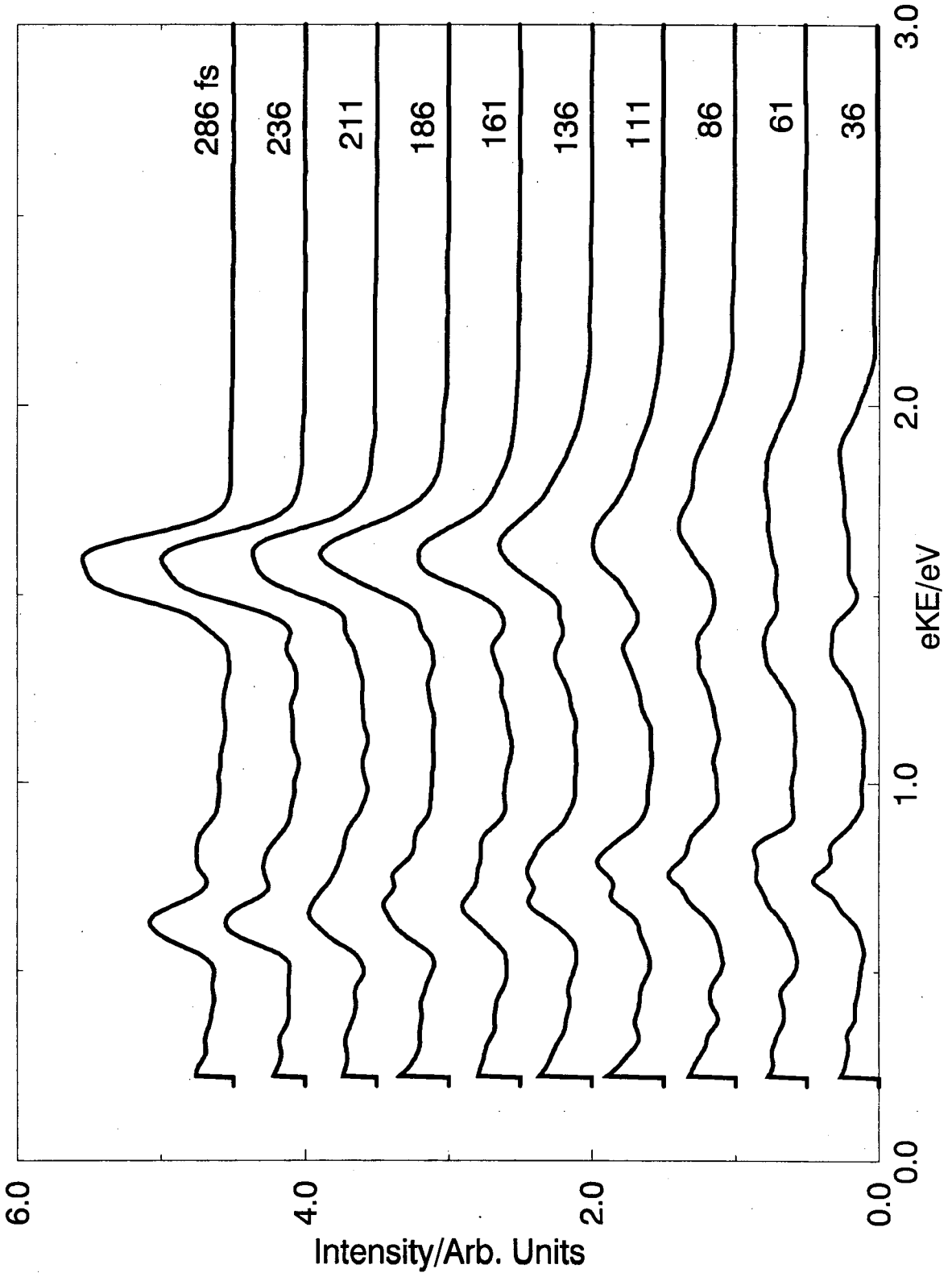

10000 fs
9000
7000
5000
4500
3500
2500
2000
1900
1800
5.5
3.5
1.5
-0.5
0.0
1.0
2.0
3.0
eKE/eV
Intensity/Arb. Units

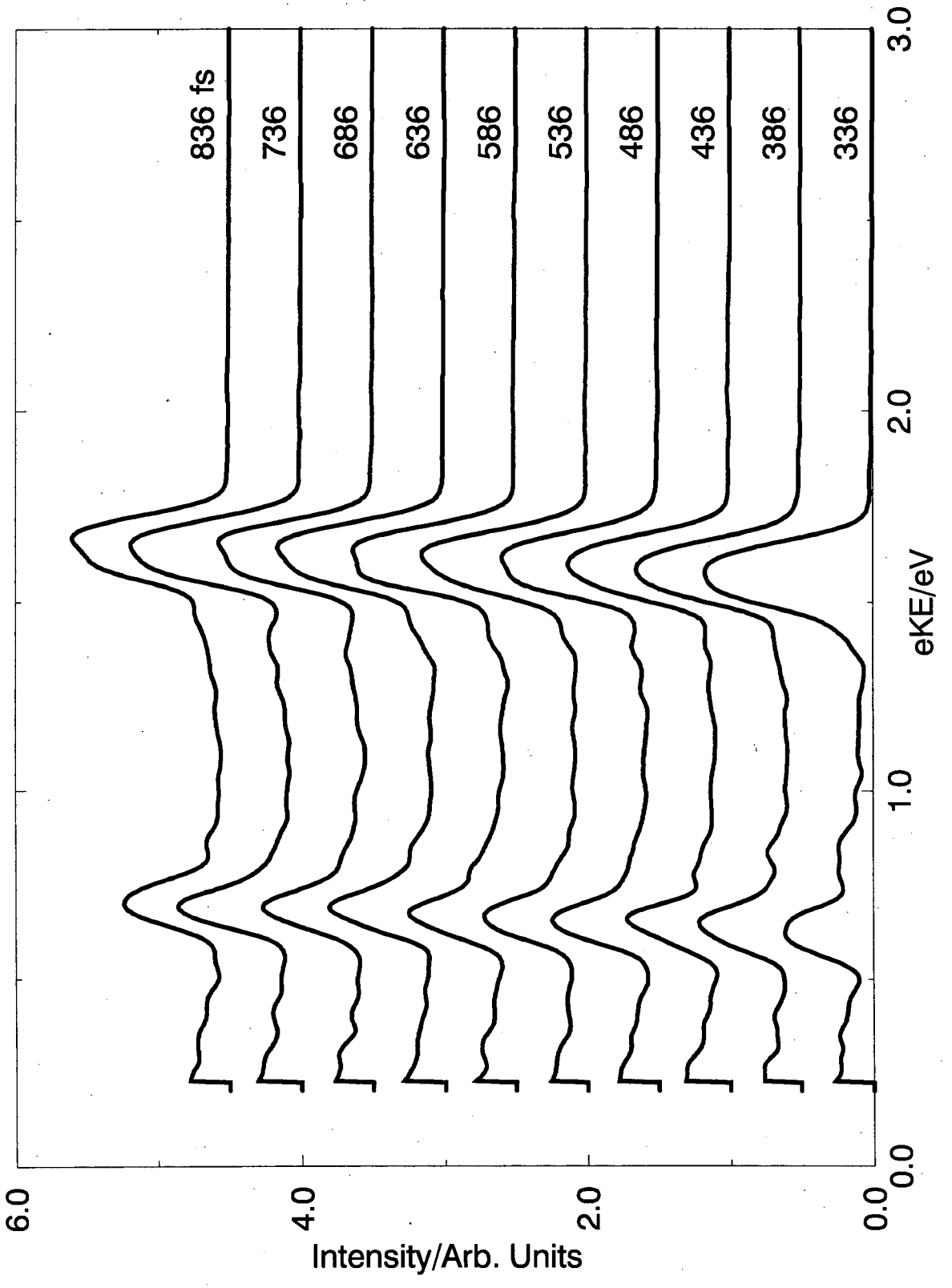

45000 fs
40000
35000
30000
25000
20000
17000
15000
13000
11000
5.5
3.5
1.5
-0.5
Intensity/Arb. Units
0.0
1.0
2.0
3.0
eKE/eV

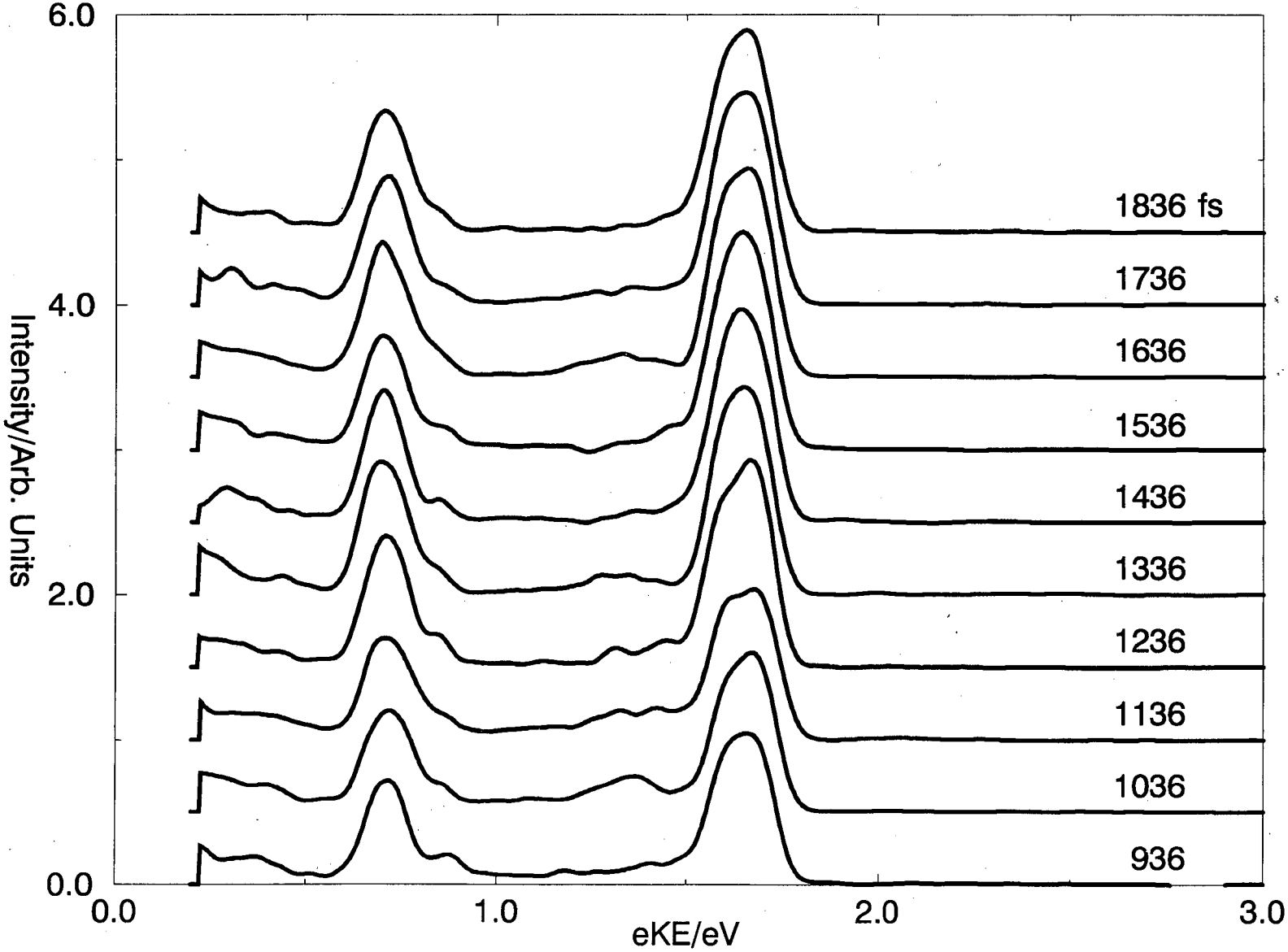

50000 fs
3.0
2.0
1.0
0.0
-0.5
eKE/eV
5.5
3.5
1.5
Intensity/Arb. Units

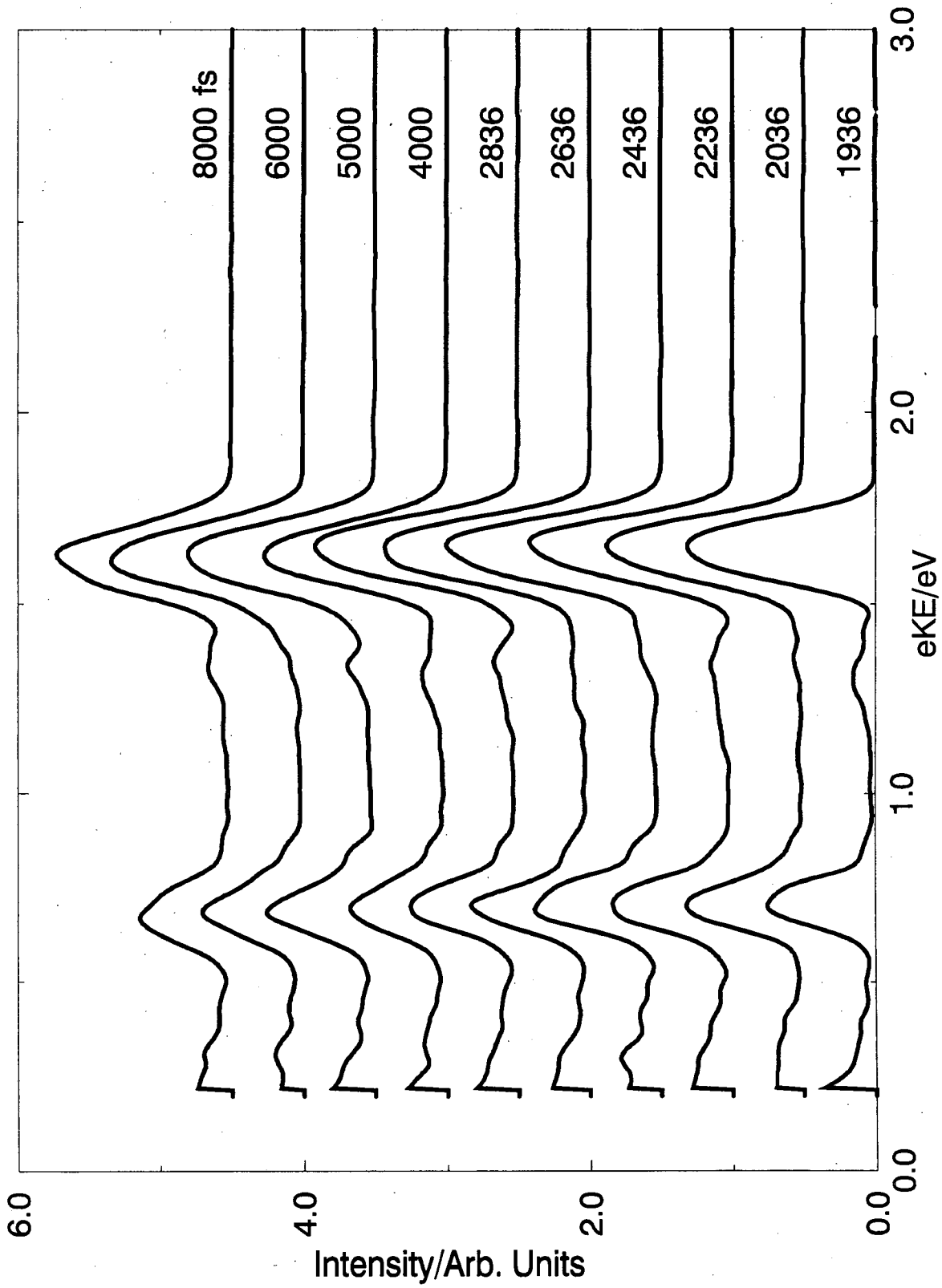
3. $I_2^-(Ar)_9$

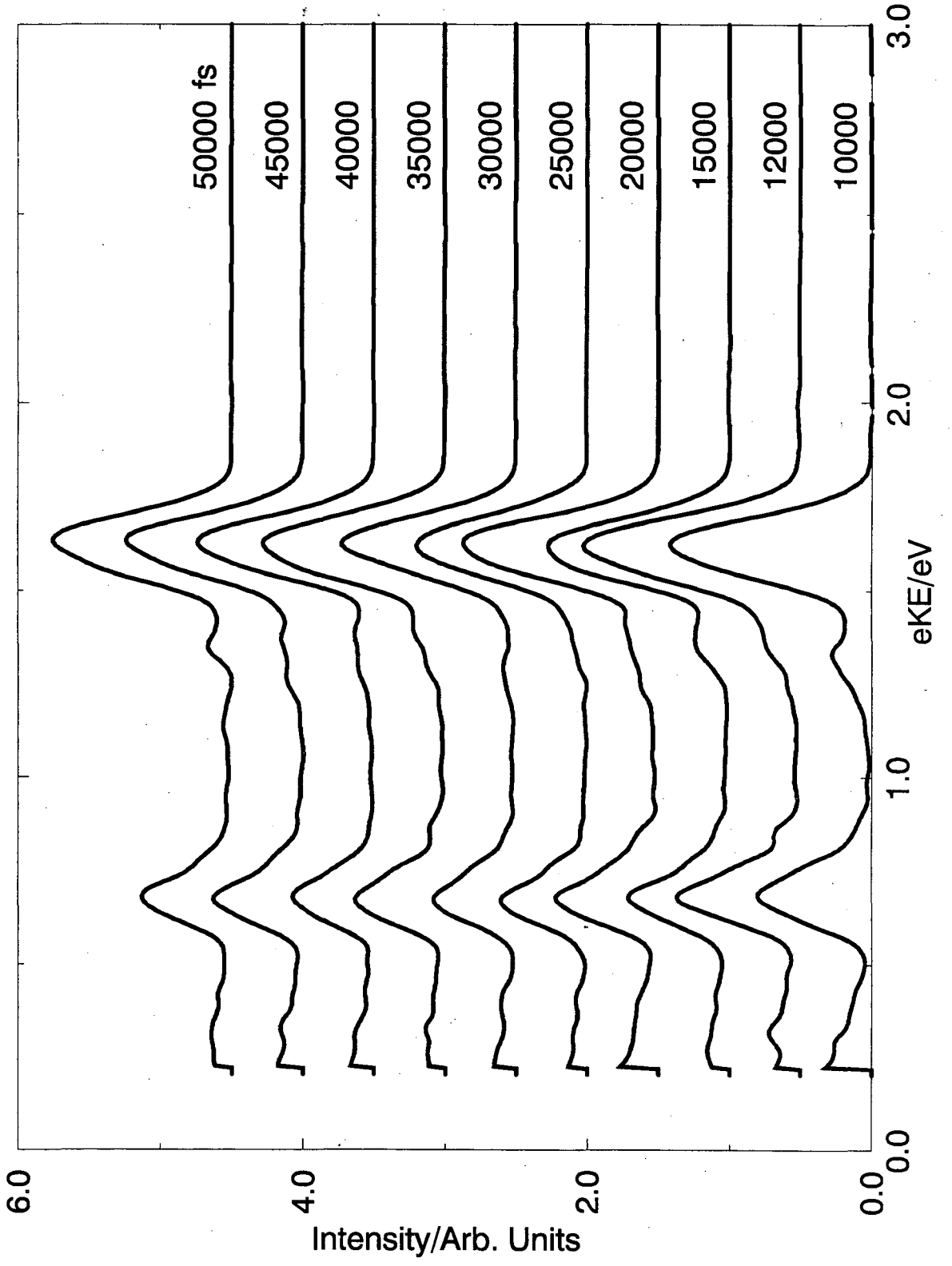

11 fs
-14
-39
-64
-89
-114
-139
-164
-214
-264
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
1.0
2.0
3.0
eKE/eV

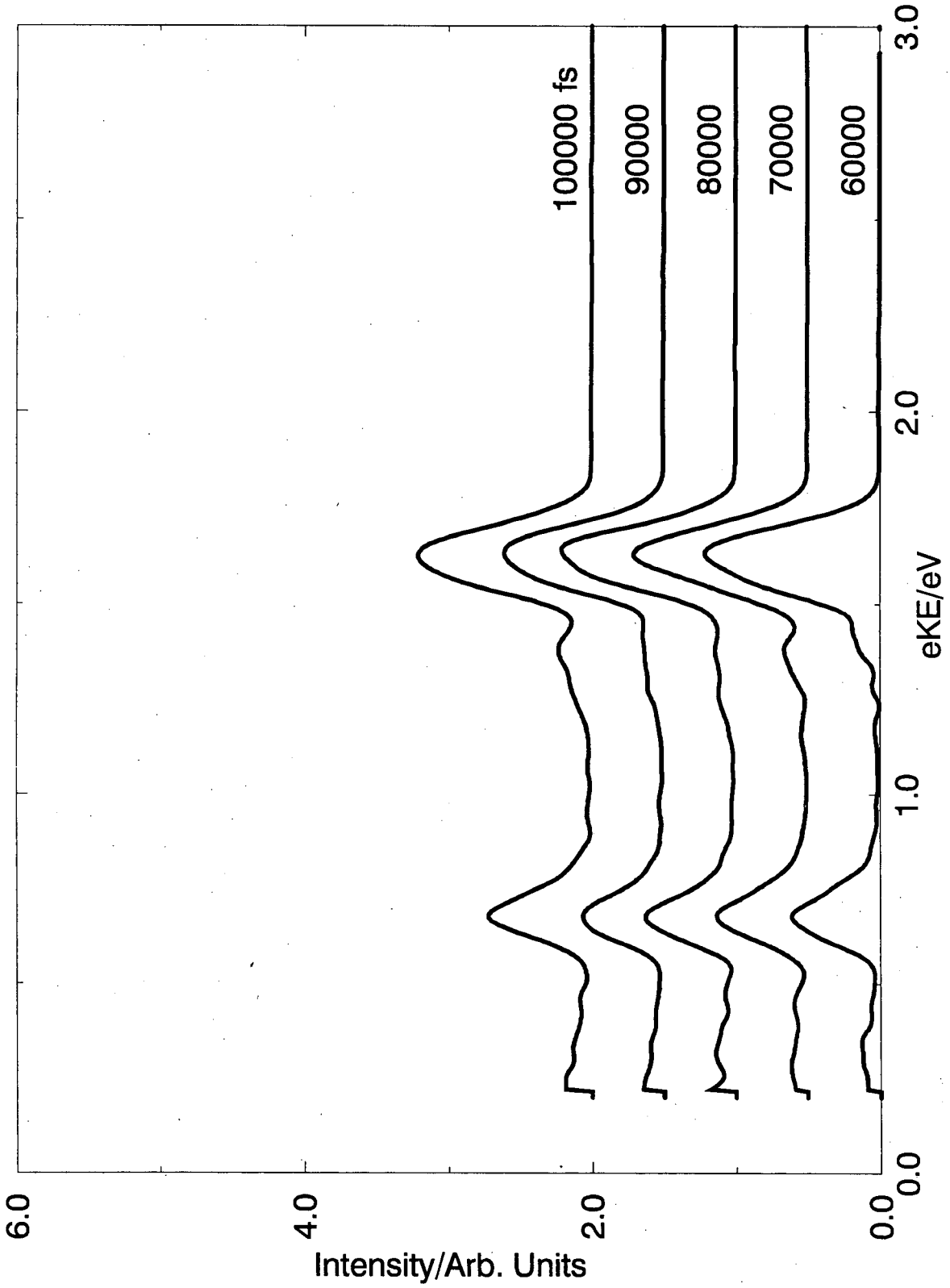

286 fs
236
211
186
161
136
111
86
61
36
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
1.0
2.0
3.0
eKE/eV


836 fs
736
686
636
586
536
486
436
386
336
3.0
2.0
1.0
0.0
eKE/eV
6.0
4.0
2.0
0.0
Intensity/Arb. Units

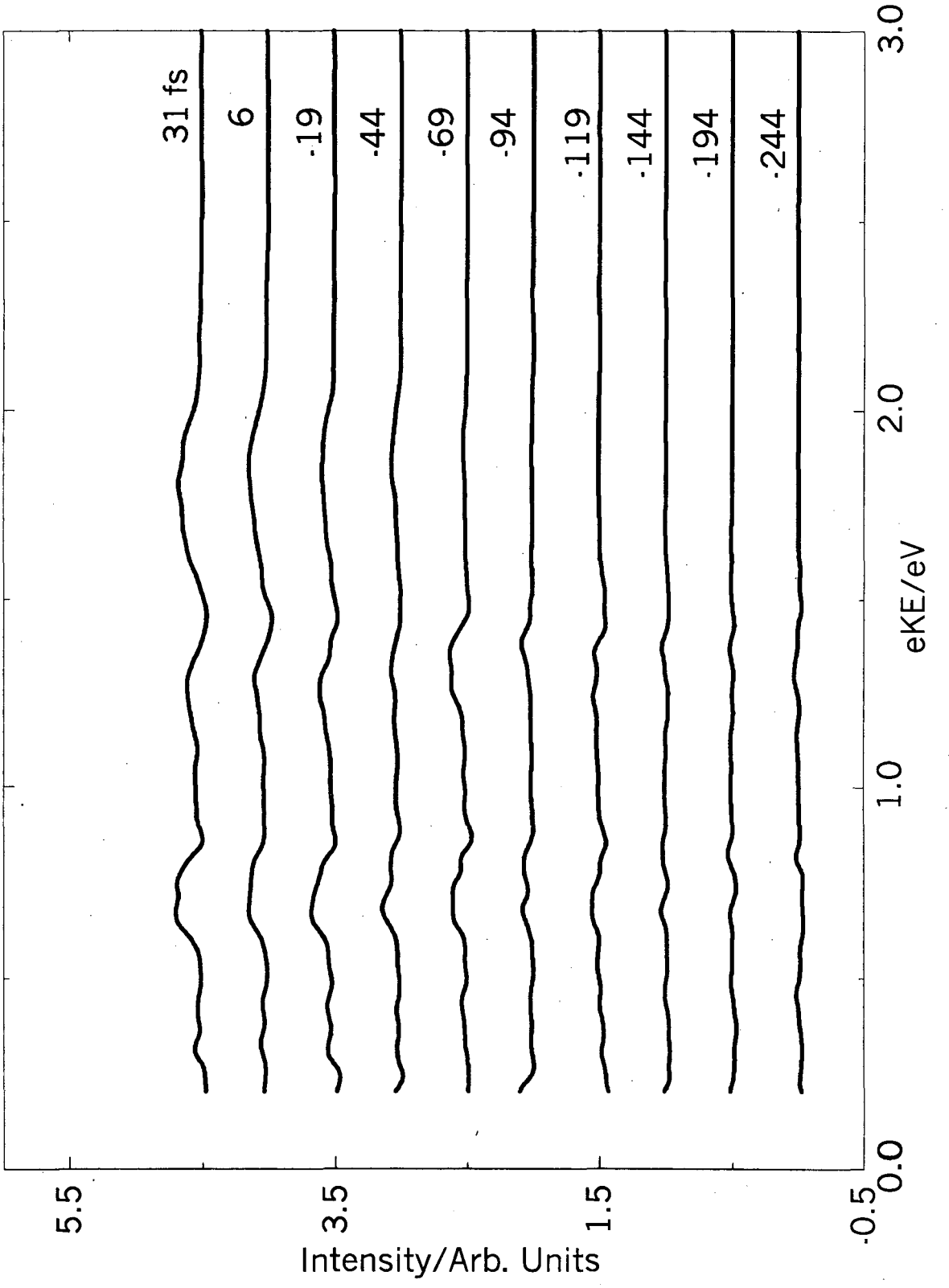

6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
1.0
2.0
3.0
eKE/eV
1836 fs
1736
1636
1536
1436
1336
1236
1136
1036
936

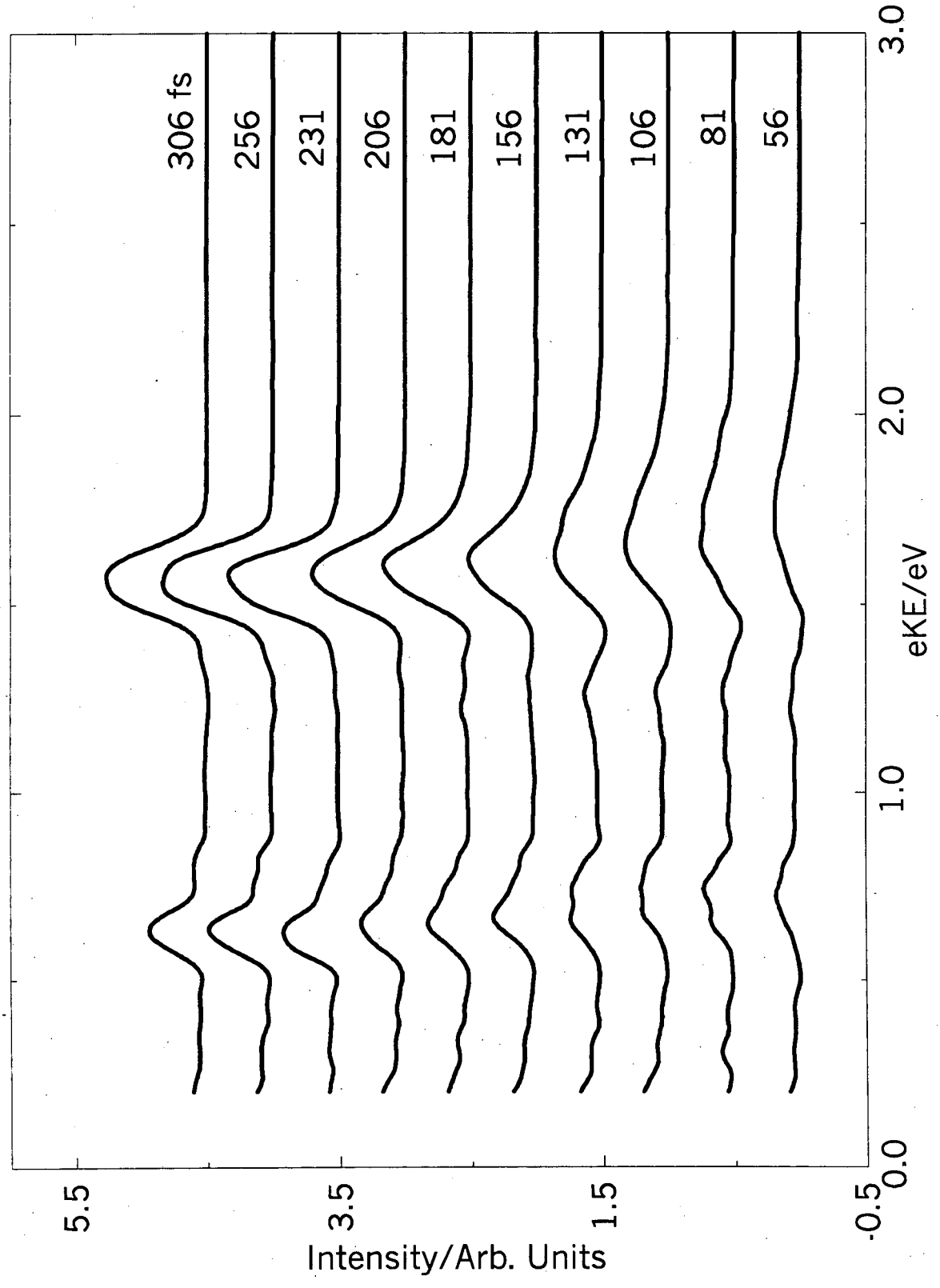

8000 fs
6000
5000
4000
2836
2636
2436
2236
2036
1936
0.0
1.0
2.0
3.0
eKE/eV
0.0
2.0
4.0
6.0
Intensity/Arb. Units


50000 fs
45000
40000
35000
30000
25000
20000
15000
12000
10000
0.0
1.0
2.0
3.0
eKE/eV
6.0
4.0
2.0
0.0
Intensity/Arb. Units

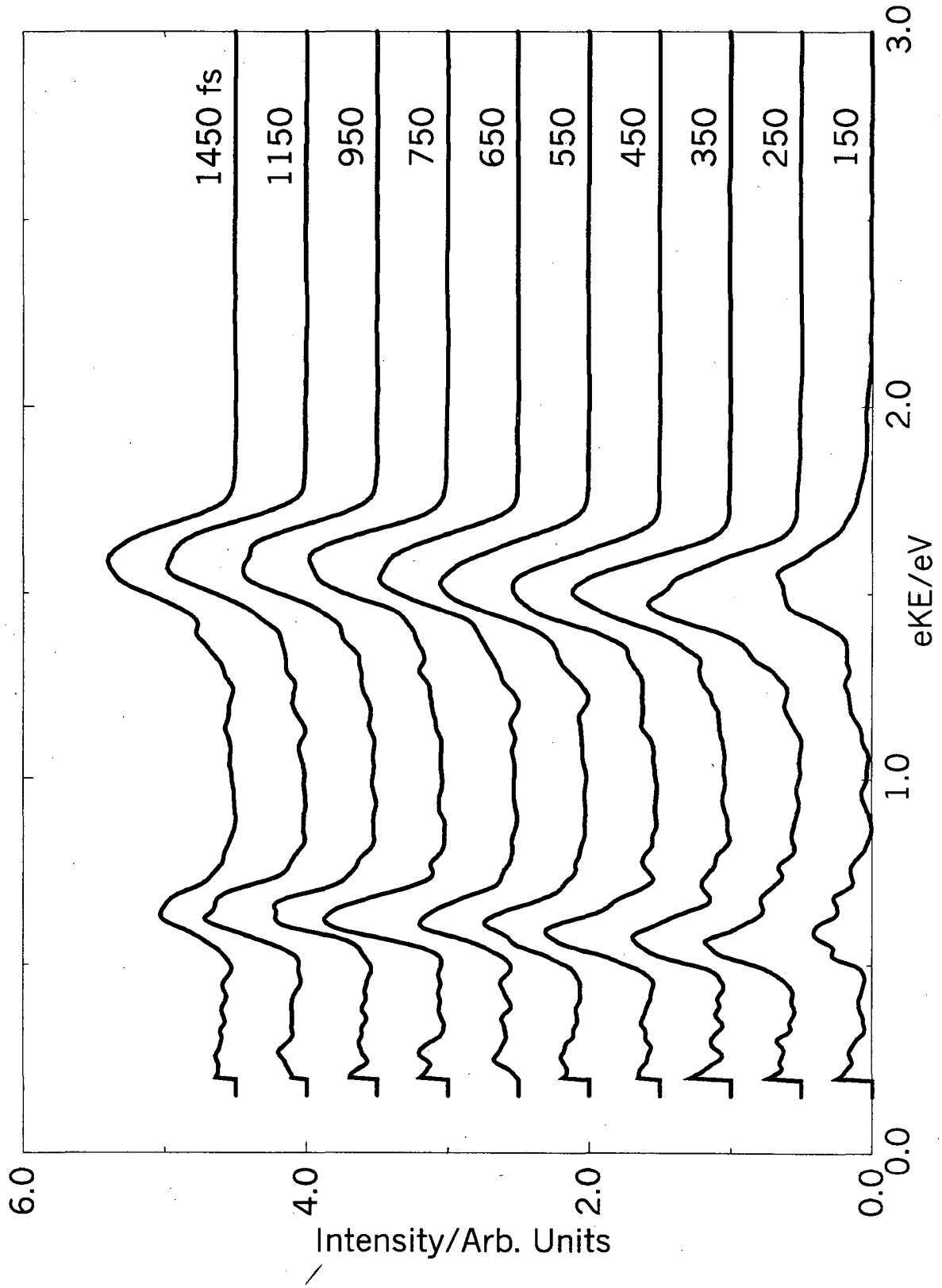

100000 fs
90000
80000
70000
60000
eKE/eV
Intensity/Arb. Units
0.0
1.0
2.0
3.0
0.0
2.0
4.0
6.0

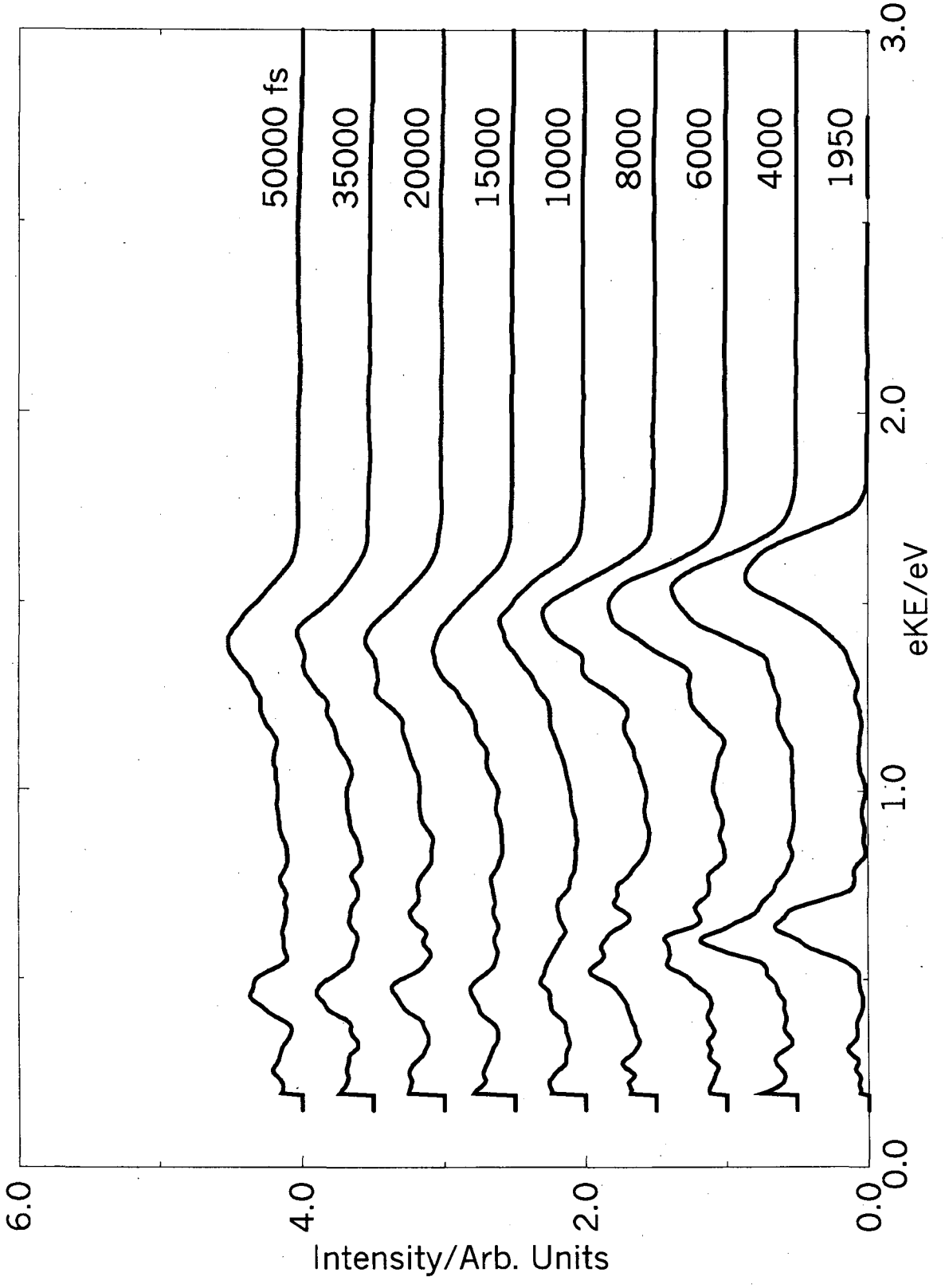
4. $I_2^-(Ar)_{12}$


31 fs
6
-19
-44
-69
-94
-119
-144
-194
-244
3.0
2.0
1.0
0.0
eKE/eV
5.5
3.5
1.5
-0.5
Intensity/Arb. Units

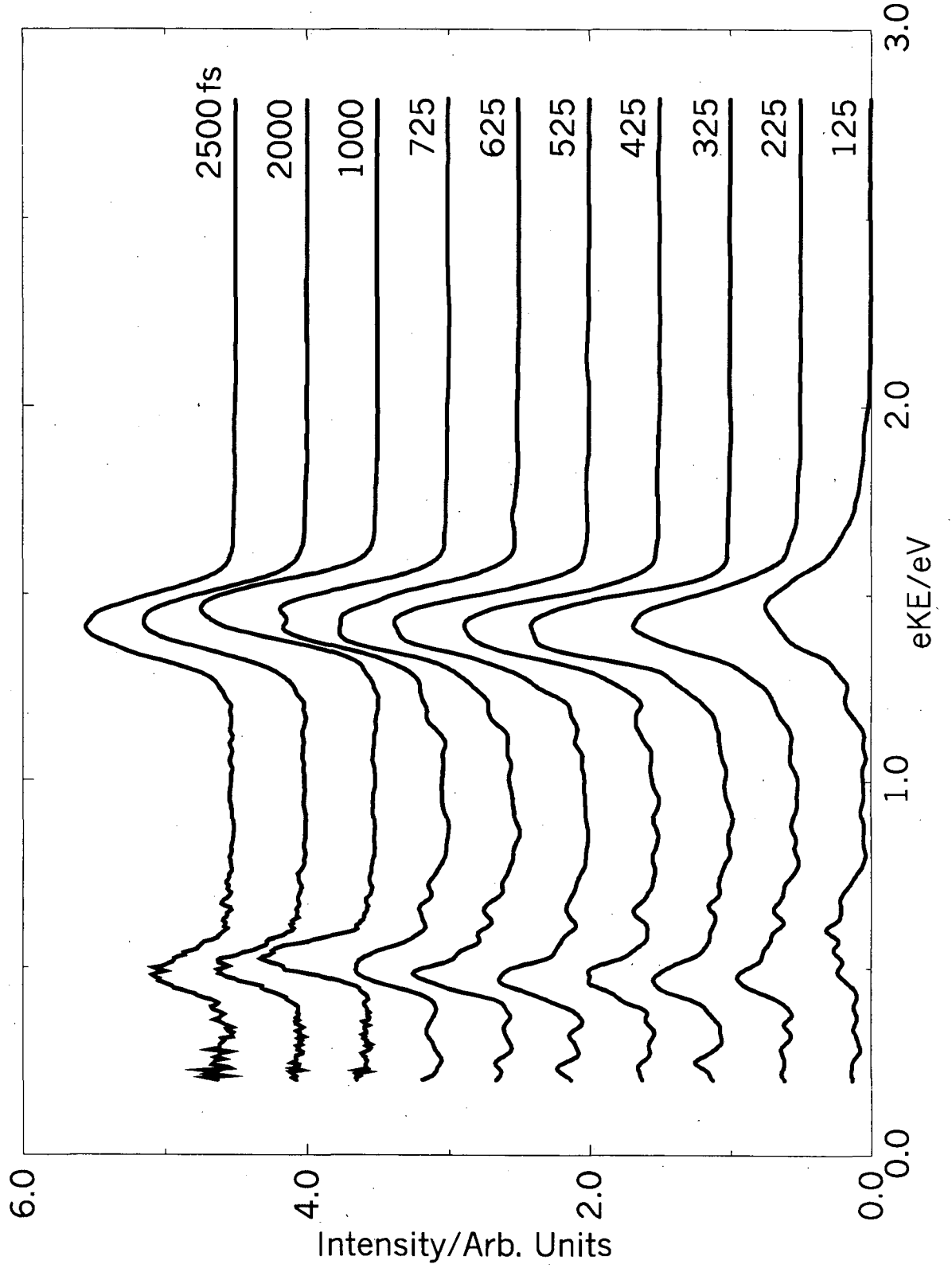

306 fs
256
231
206
181
156
131
106
81
56
3.0
2.0
1.0
0.0
eKE/eV
5.5
3.5
1.5
-0.5
Intensity/Arb. Units

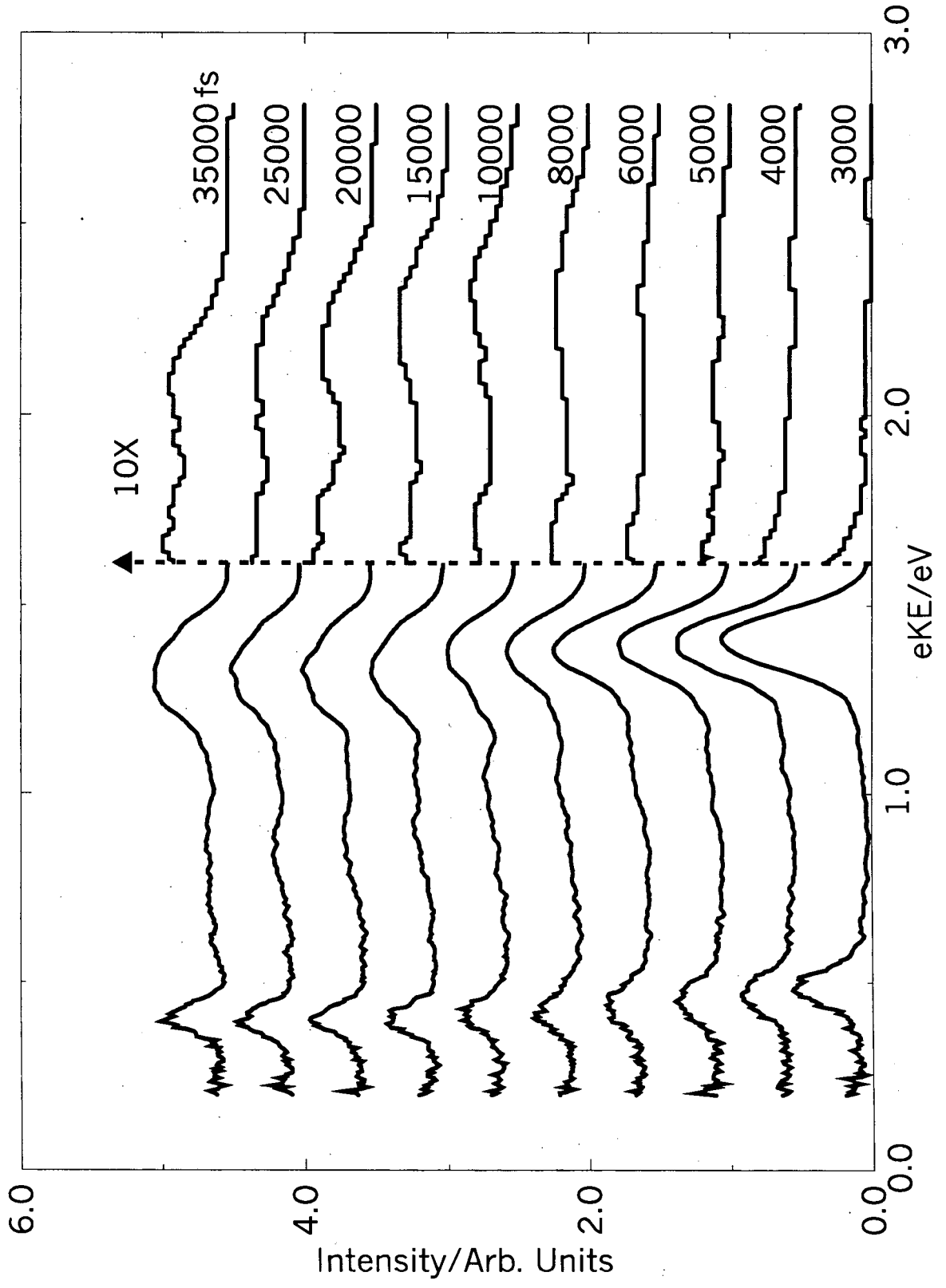
5. $I_2^-(Ar)_{16}$

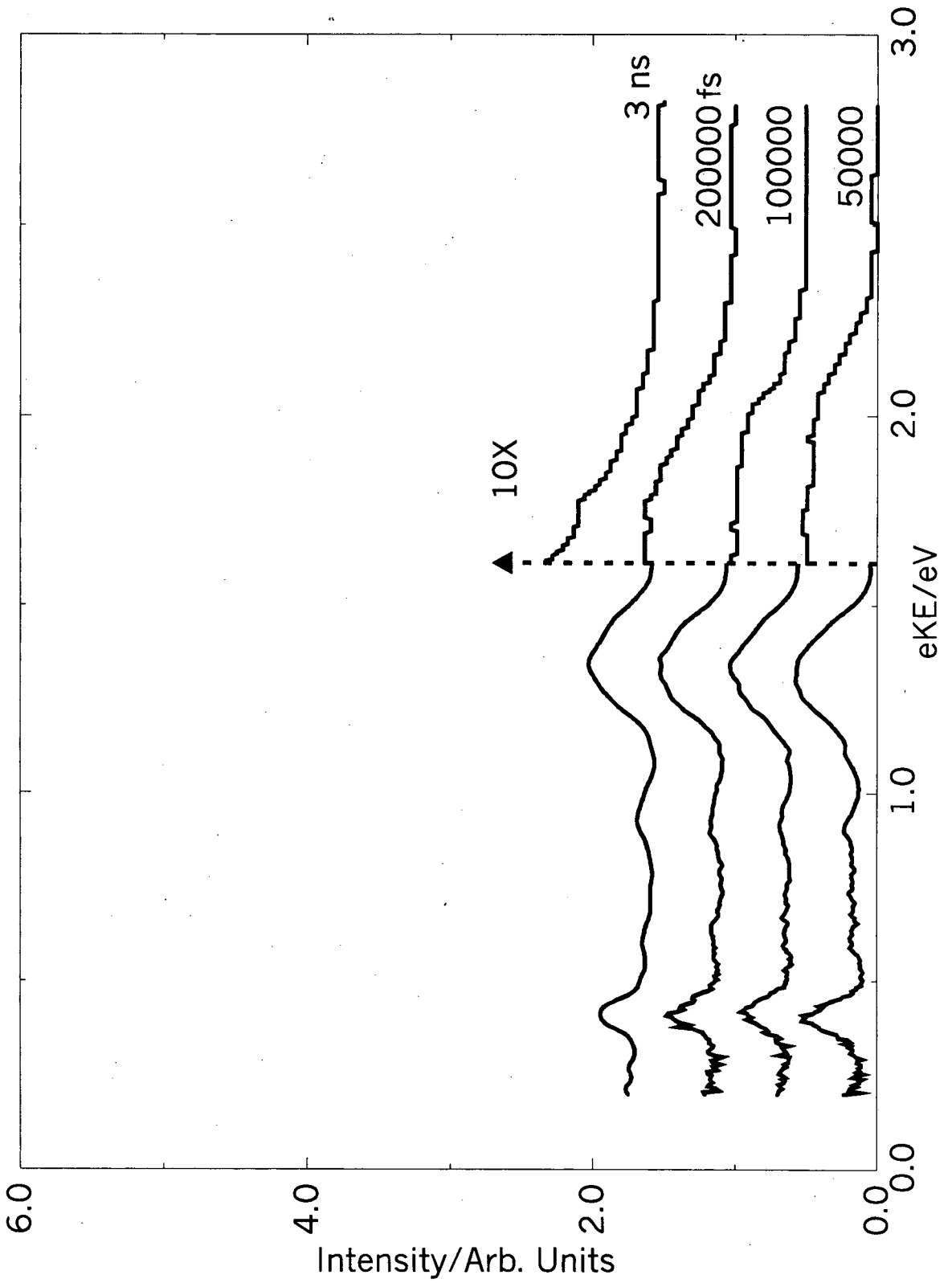

1450 fs
1150
950
750
650
550
450
350
250
150
3.0
2.0
1.0
0.0
eKE/eV
6.0
4.0
2.0
0.0
Intensity/Arb. Units


50000 fs
35000
20000
15000
10000
8000
6000
4000
1950
0.0
1.0
2.0
3.0
eKE/eV
6.0
4.0
2.0
0.0
Intensity/Arb. Units

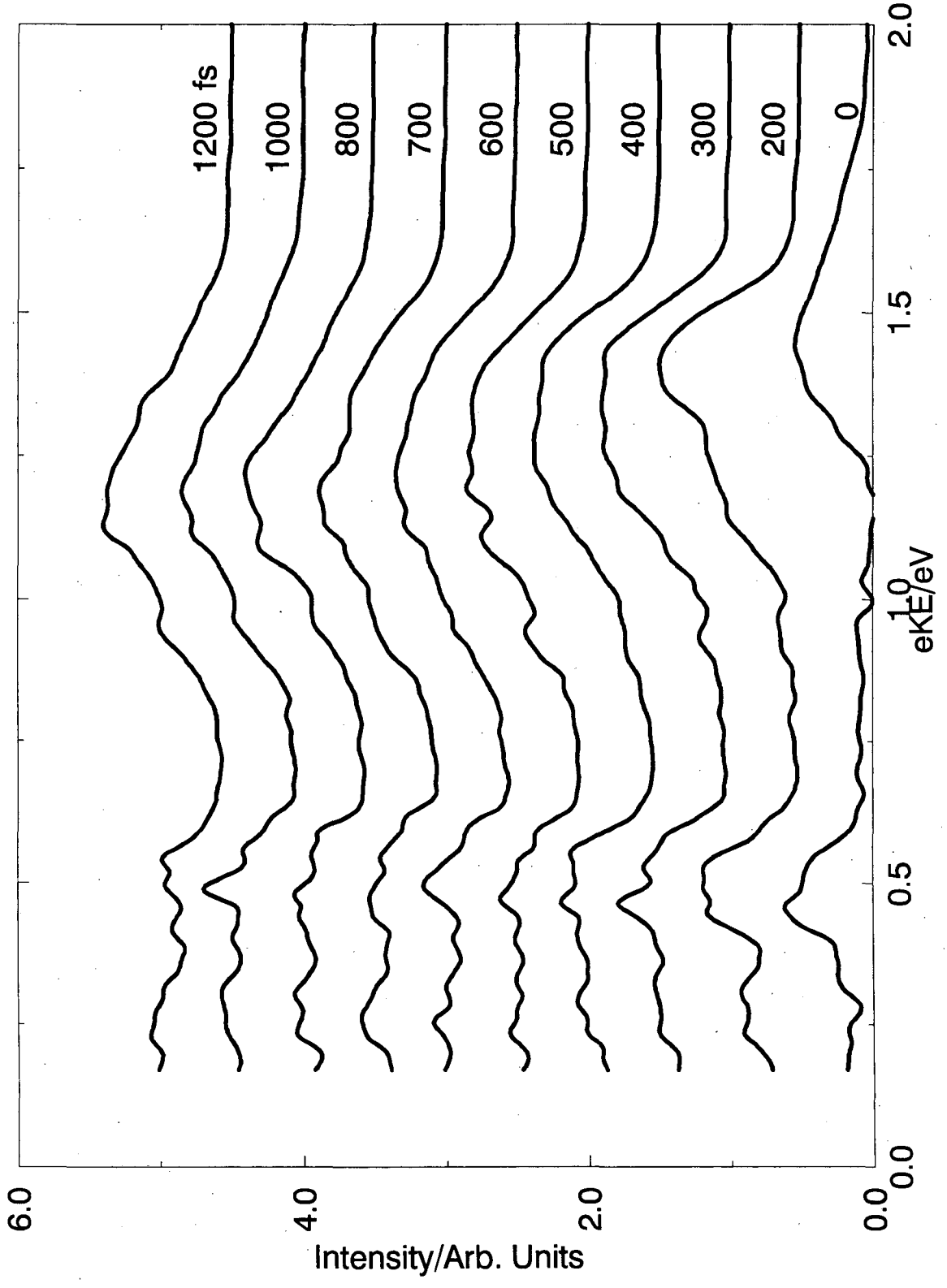
6. $I_2^-(Ar)_{20}$

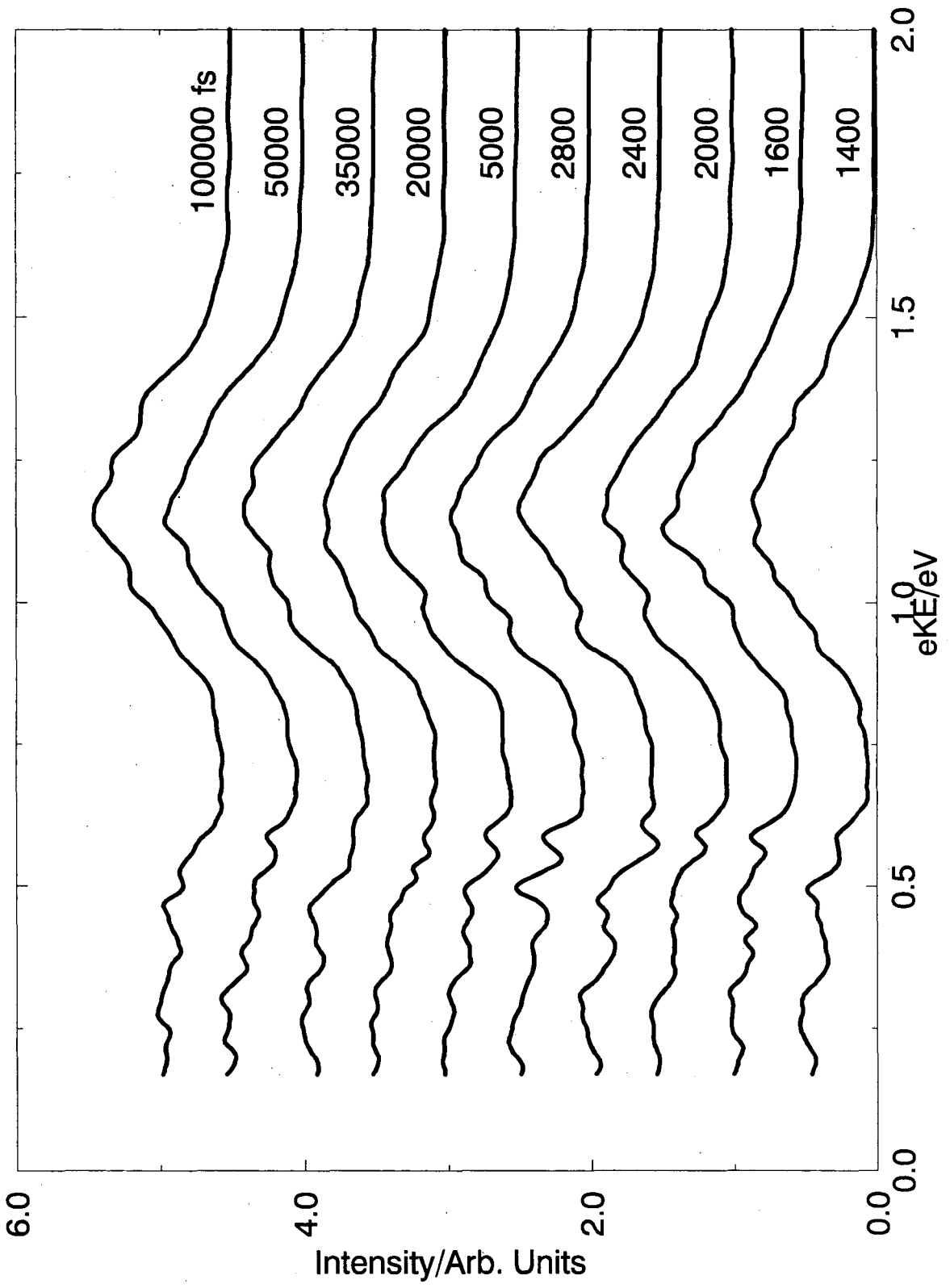

2500 fs
2000
1000
725
625
525
425
325
225
125
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
1.0
2.0
3.0
eKE/eV

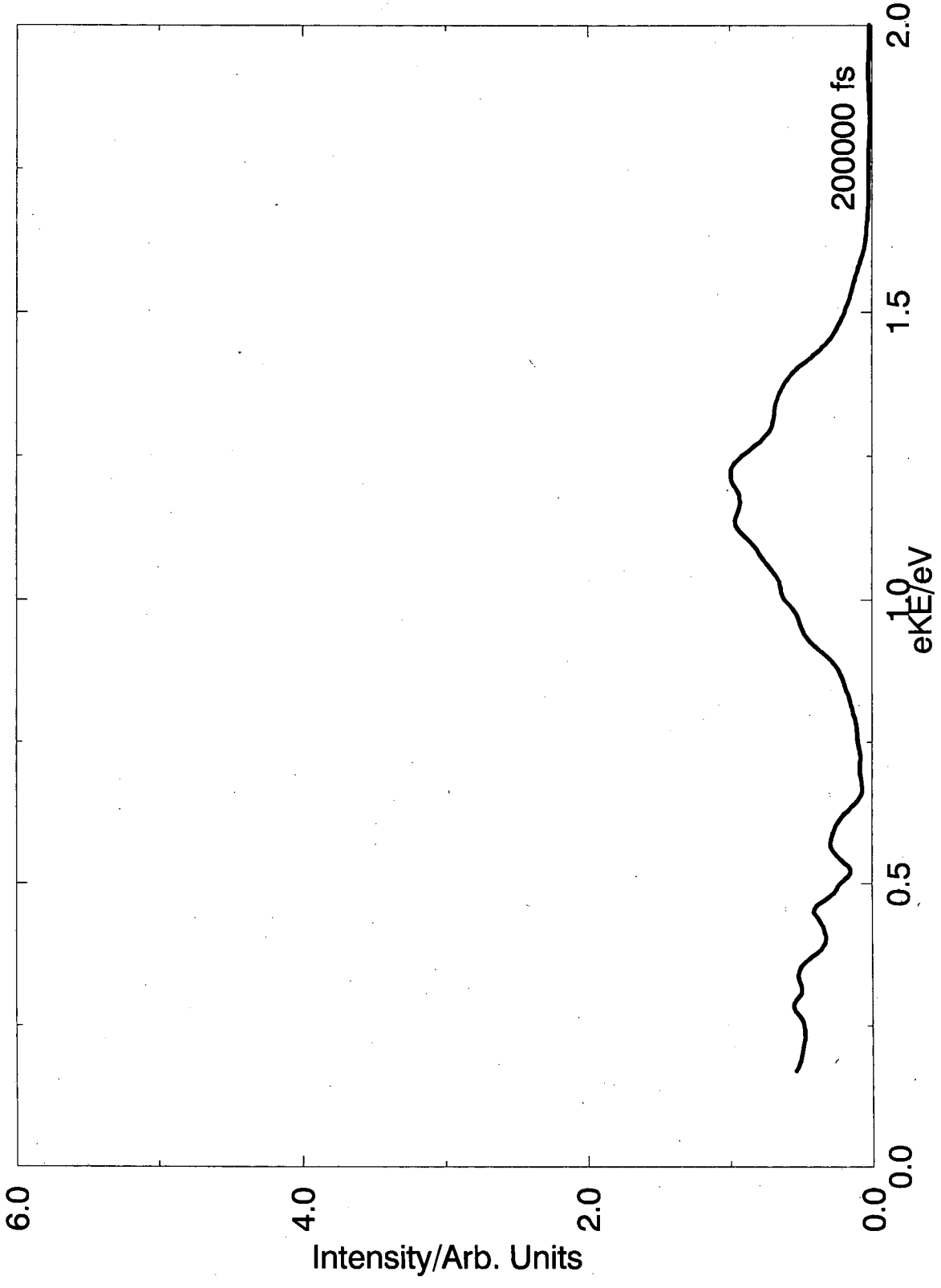

35000 fs
25000
20000
15000
10000
8000
6000
5000
4000
3000
10X
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
1.0
2.0
3.0
eKE/eV


3 ns
200000 fs
100000
50000
10X
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
1.0
2.0
3.0
eKE/eV

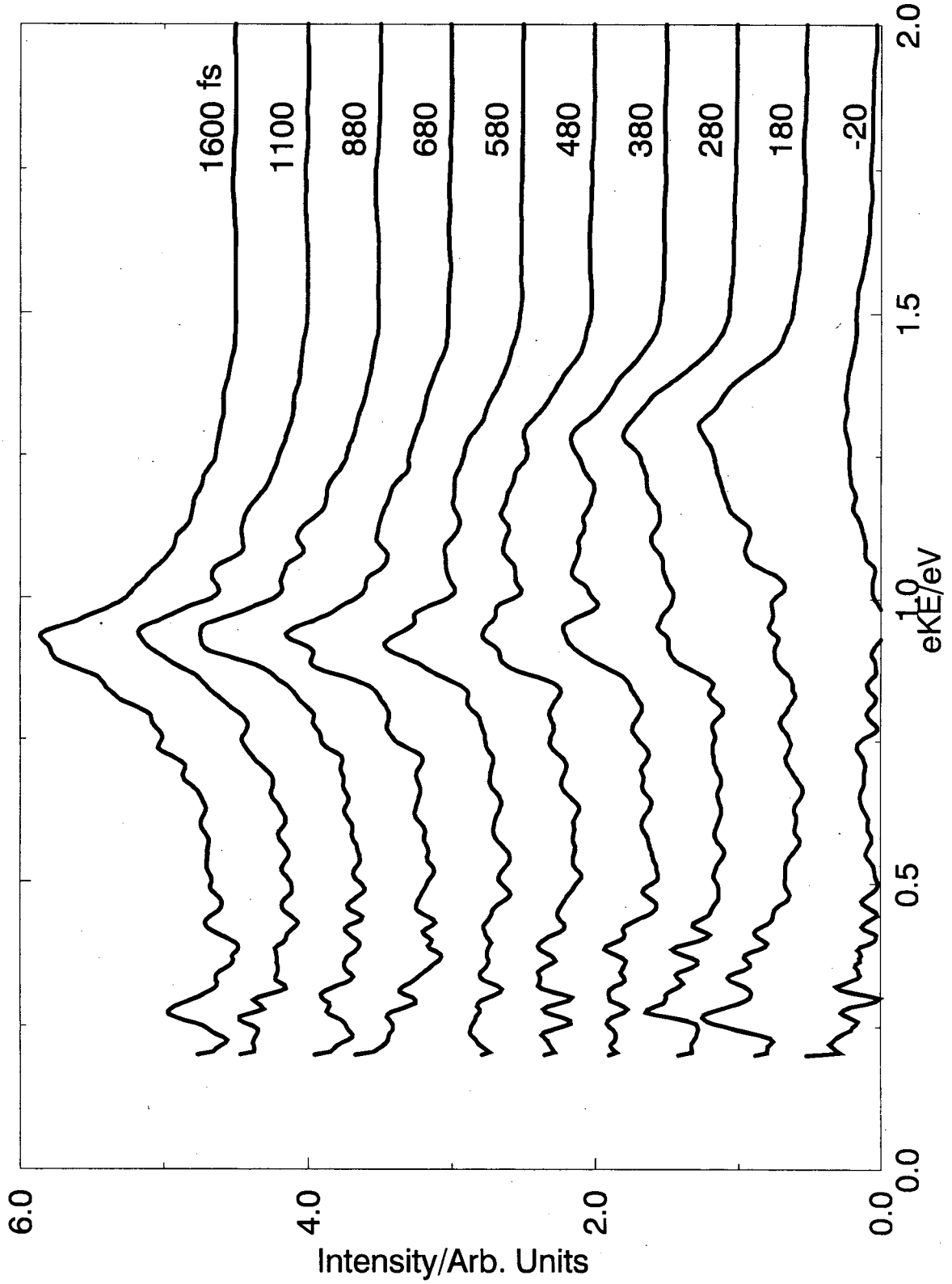
7. $I_2^-(CO_2)_4$

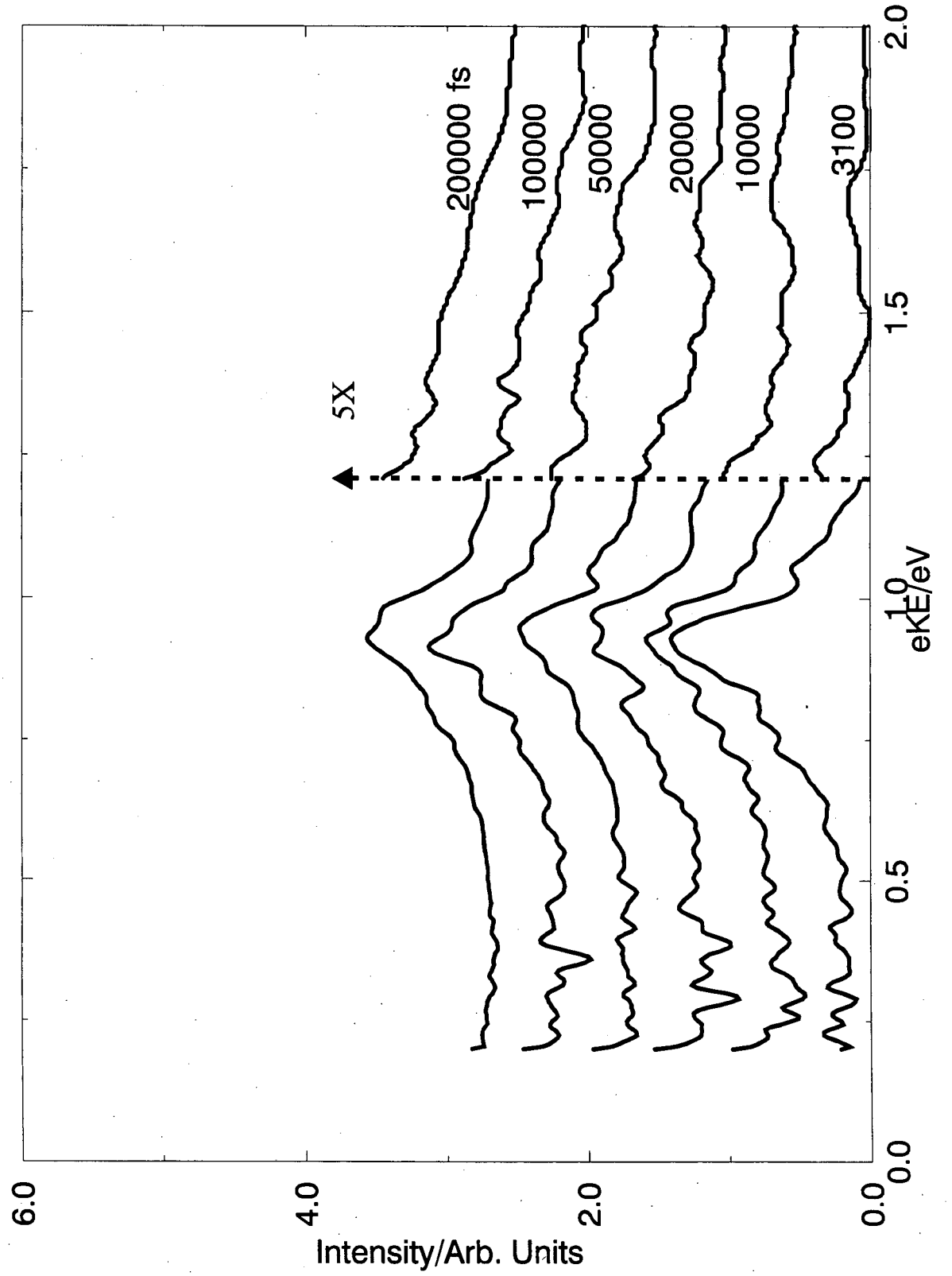

1200 fs
1000
800
700
600
500
400
300
200
0
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV


100000 fs
50000
35000
20000
5000
2800
2400
2000
1600
1400
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV

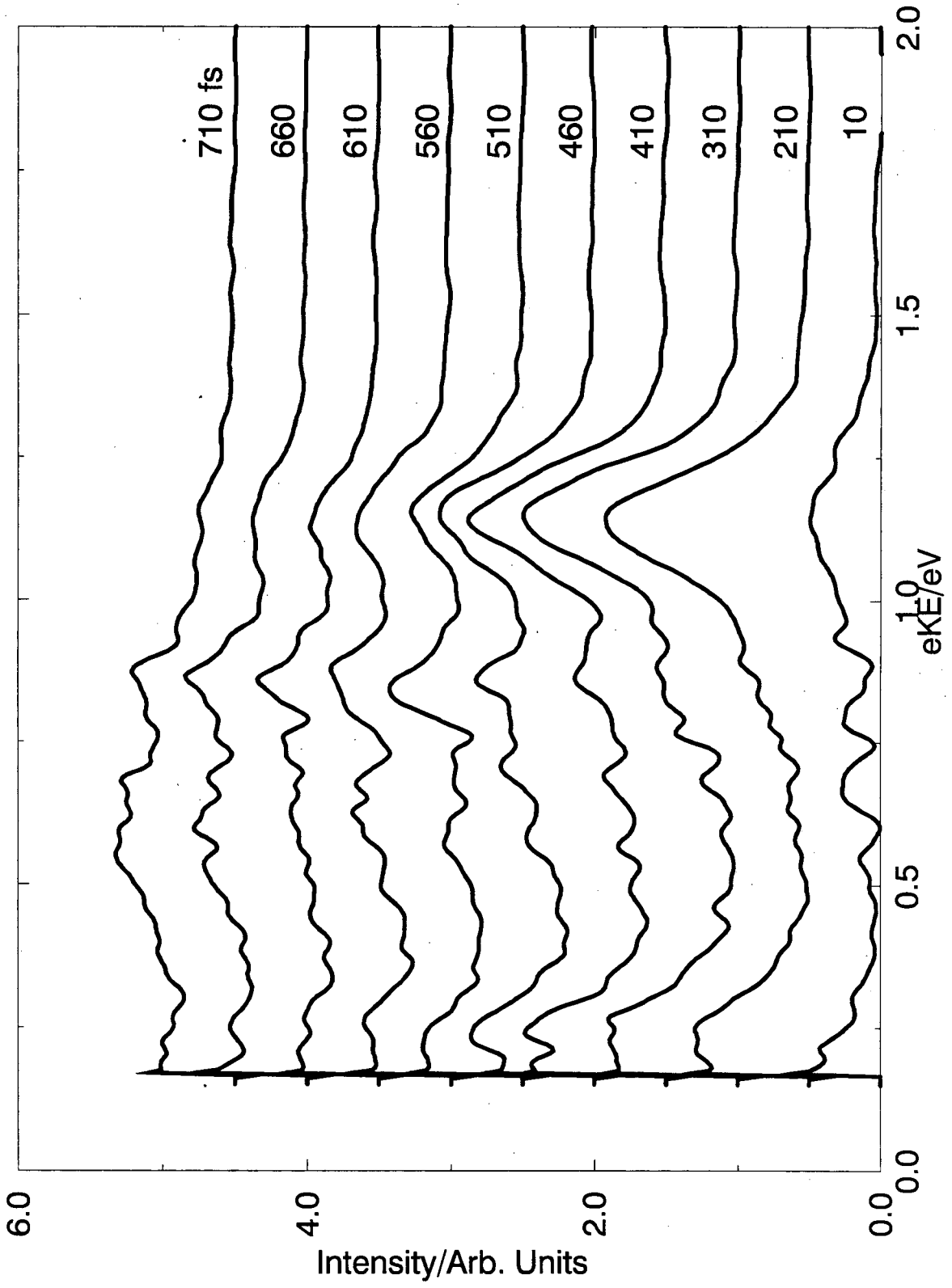

200000 fs
2.0
1.5
1.0
0.5
0.0
eKE/eV
6.0
4.0
2.0
0.0
Intensity/Arb. Units

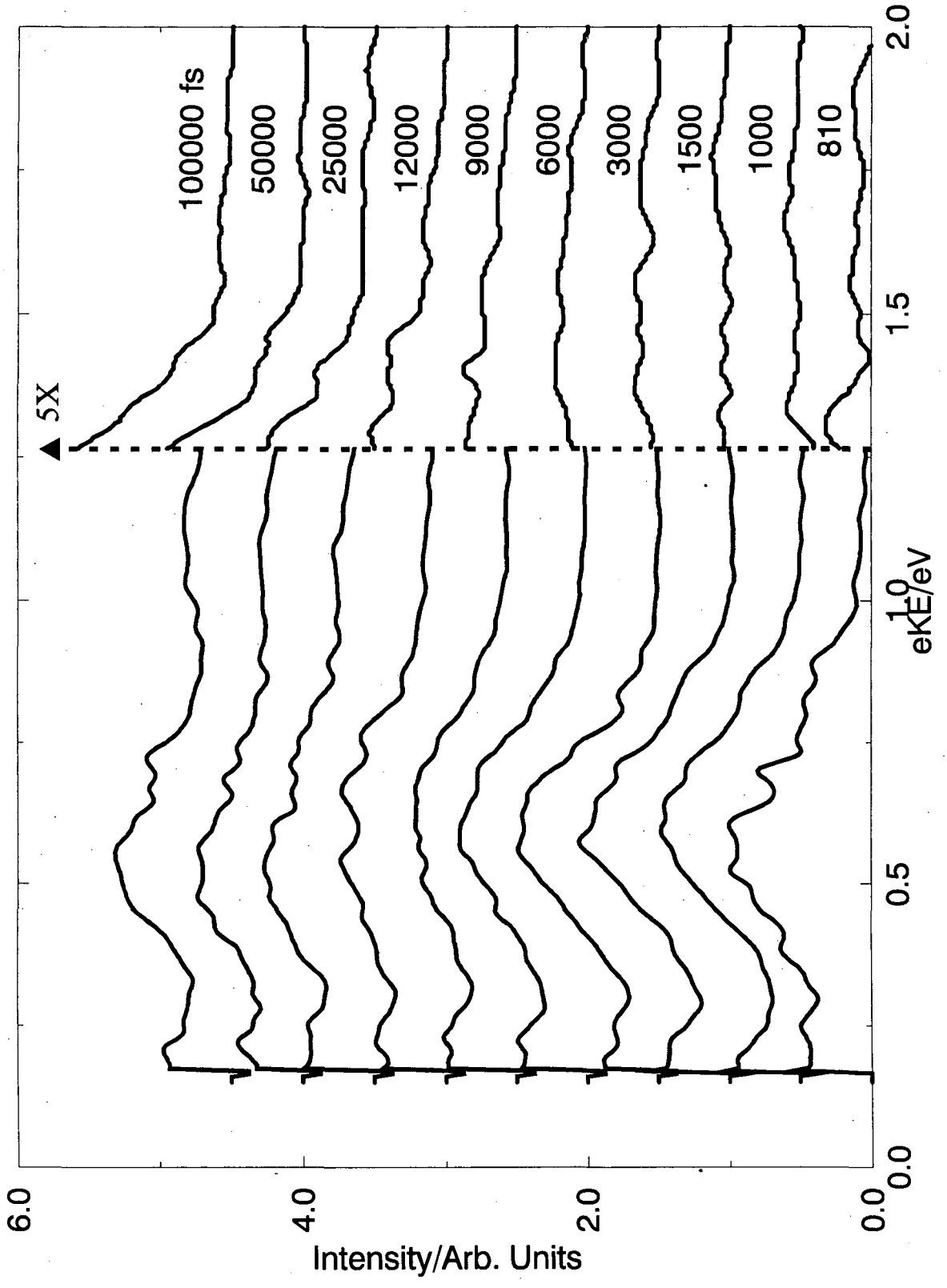
8. $I_2^-(CO_2)_6$

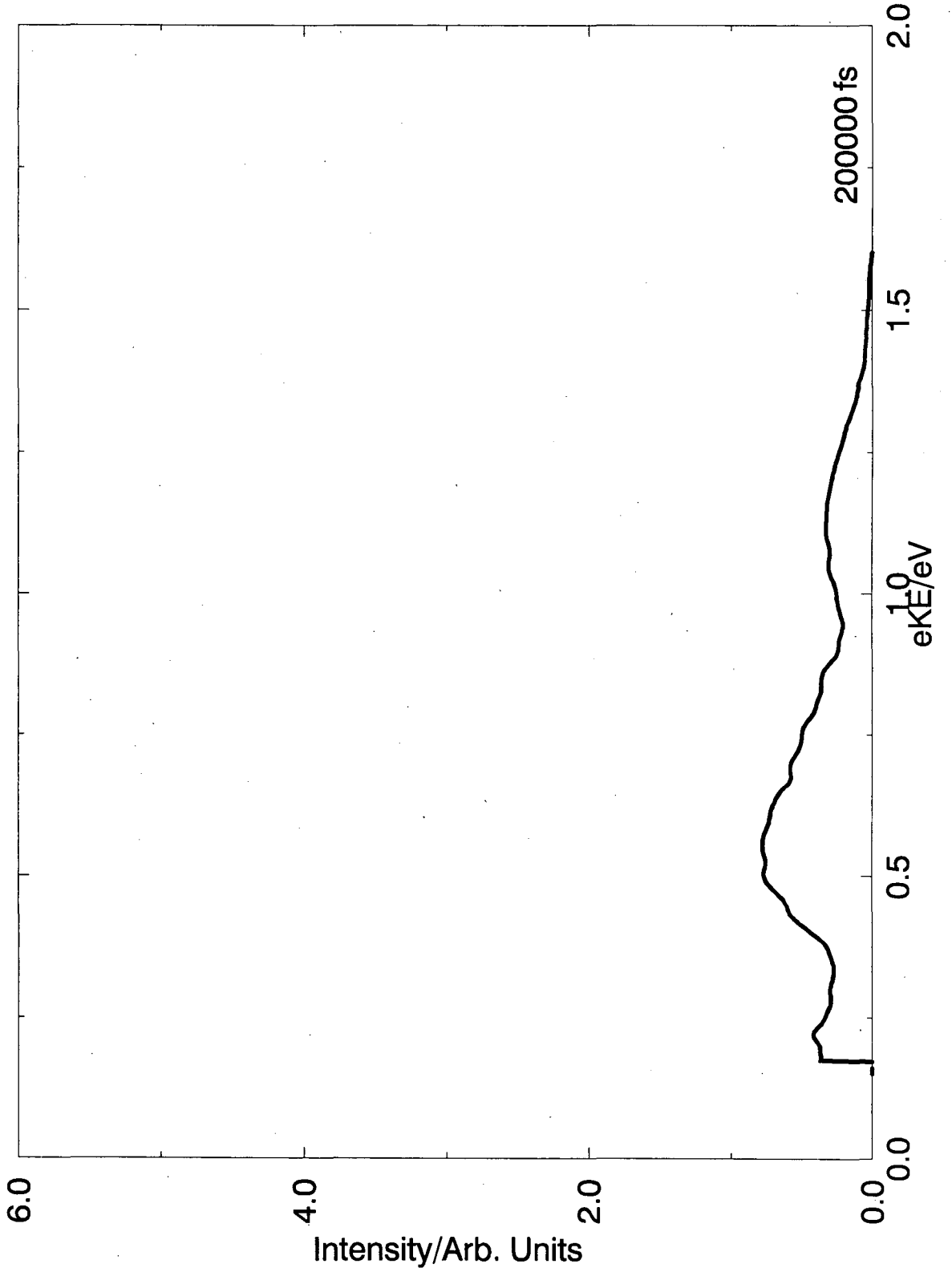

1600 fs
1100
880
680
580
480
380
280
180
-20
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV


200000 fs
100000
50000
20000
10000
3100
5X
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV

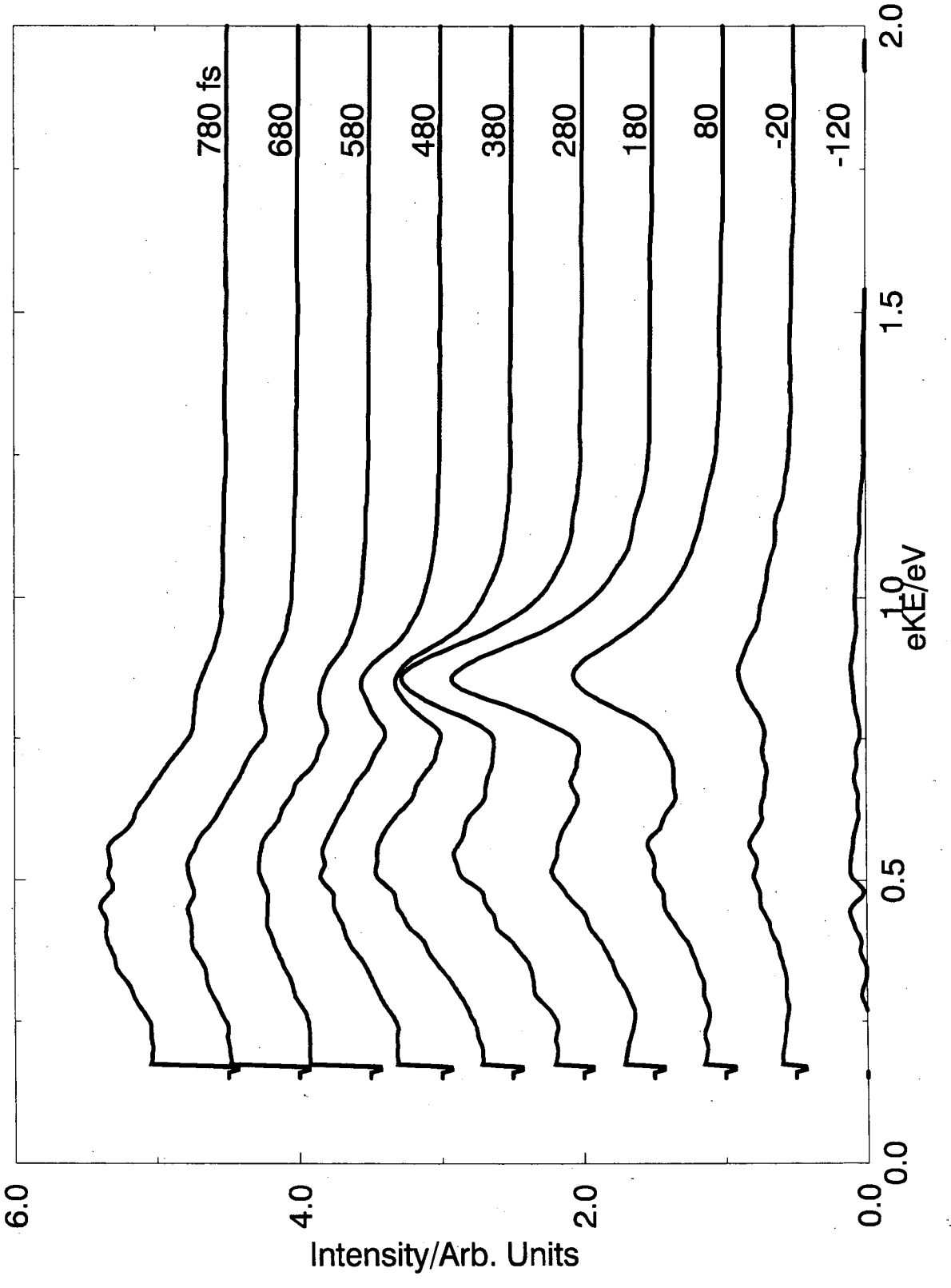
9. $I_2^-(CO_2)_9$

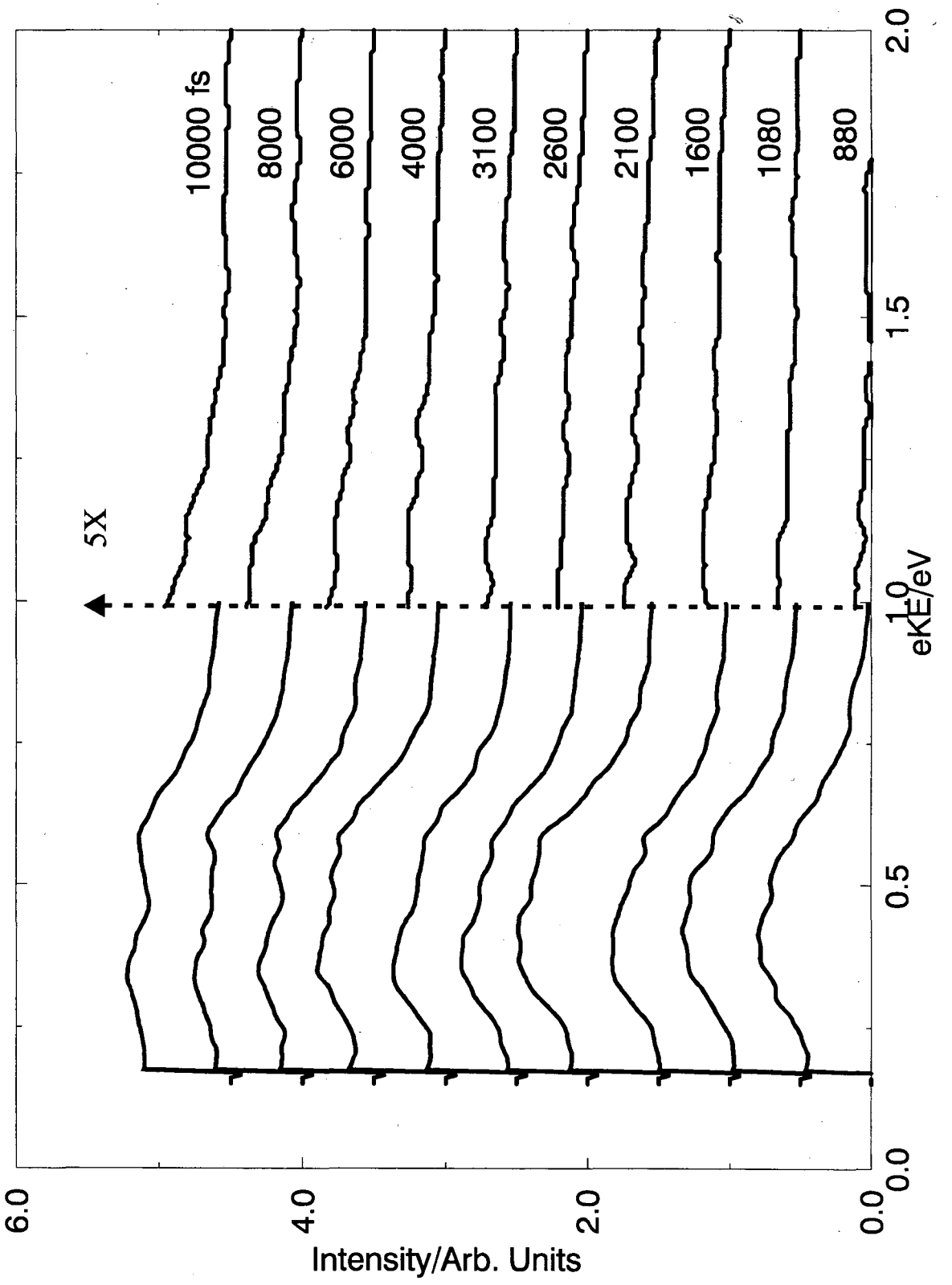

710 fs
660
610
560
510
460
410
310
210
10
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV

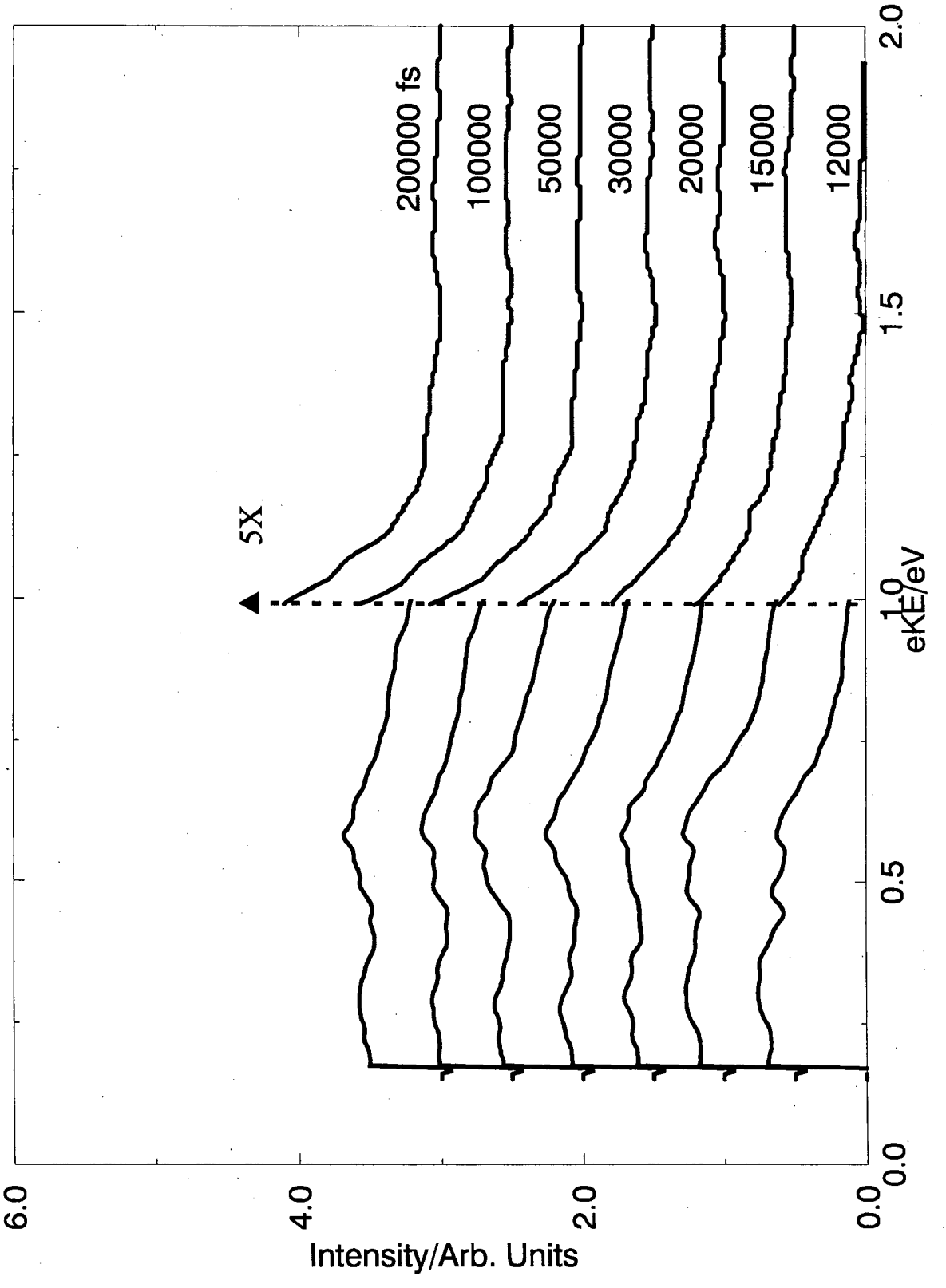

100000 fs
50000
25000
12000
9000
6000
3000
1500
1000
810
5X
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV


200000 fs
0.0
0.5
1.0
1.5
2.0
eKE/eV
6.0
4.0
2.0
0.0
Intensity/Arb. Units

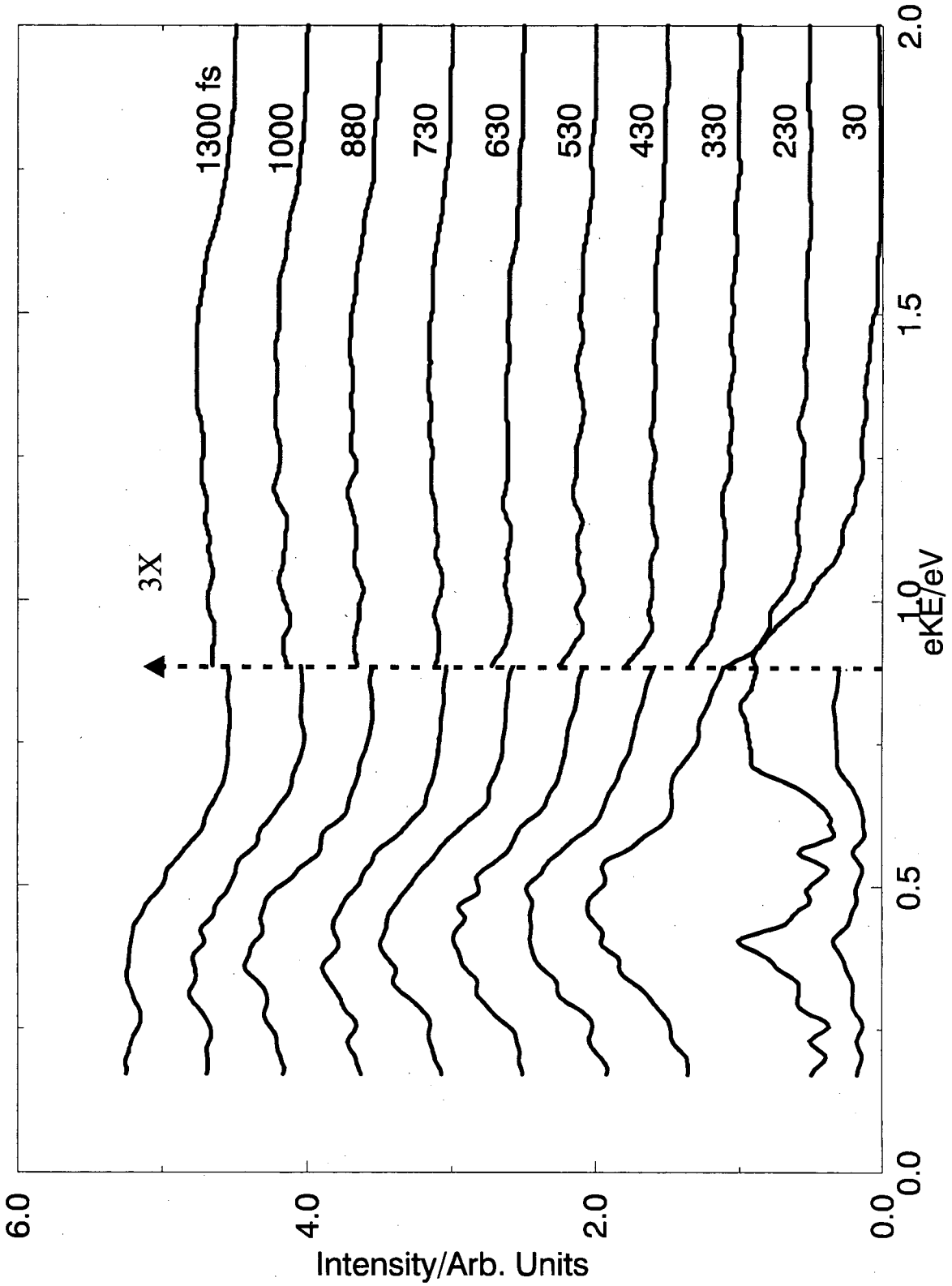
10. $I_2^-(CO_2)_{12}$

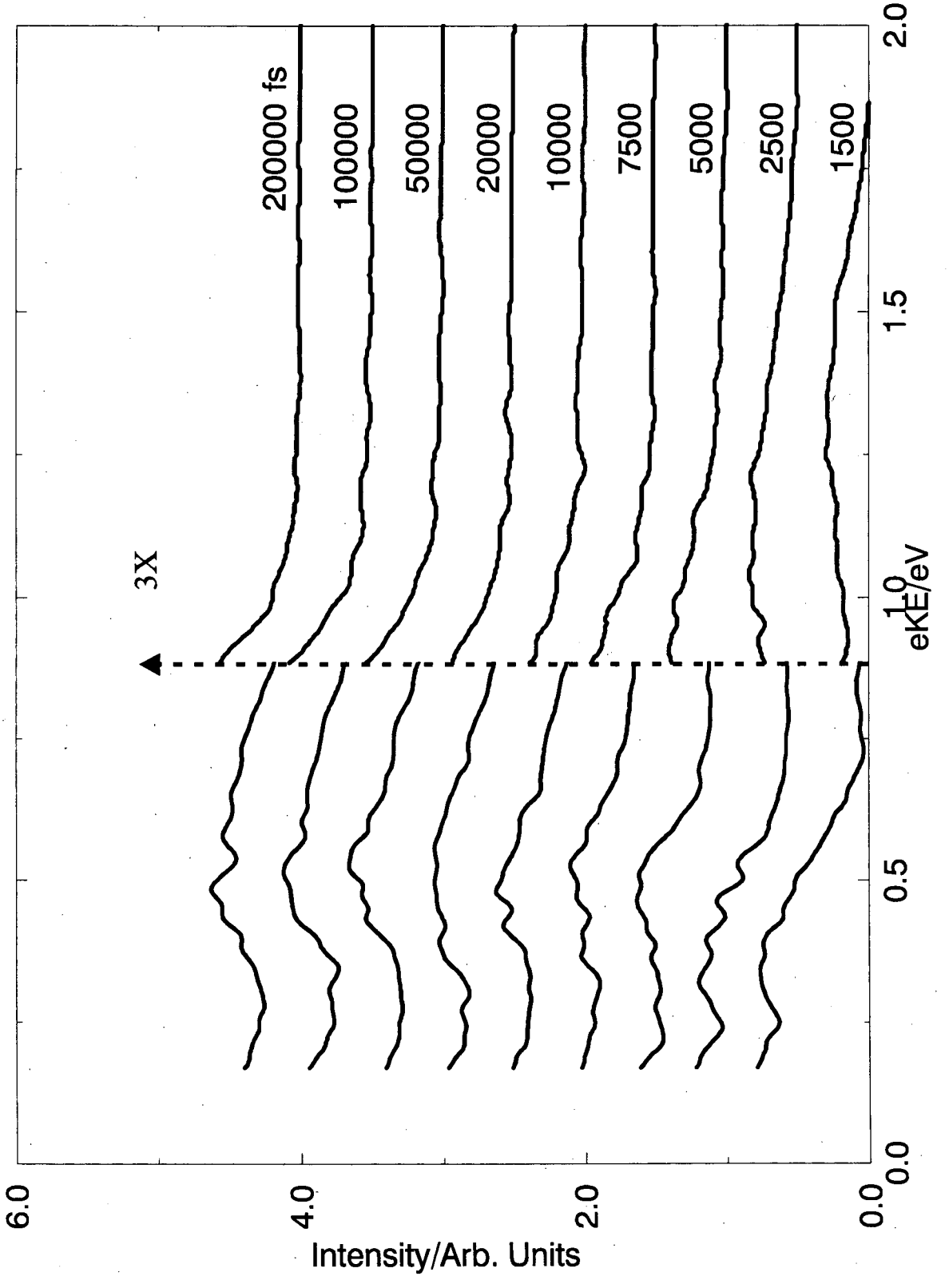

780 fs
680
580
480
380
280
180
80
-20
-120
2.0
1.5
1.0
0.5
0.0
eKE/eV
6.0
4.0
2.0
0.0
Intensity/Arb. Units


10000 fs
8000
6000
4000
3100
2600
2100
1600
1080
880
5X
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV

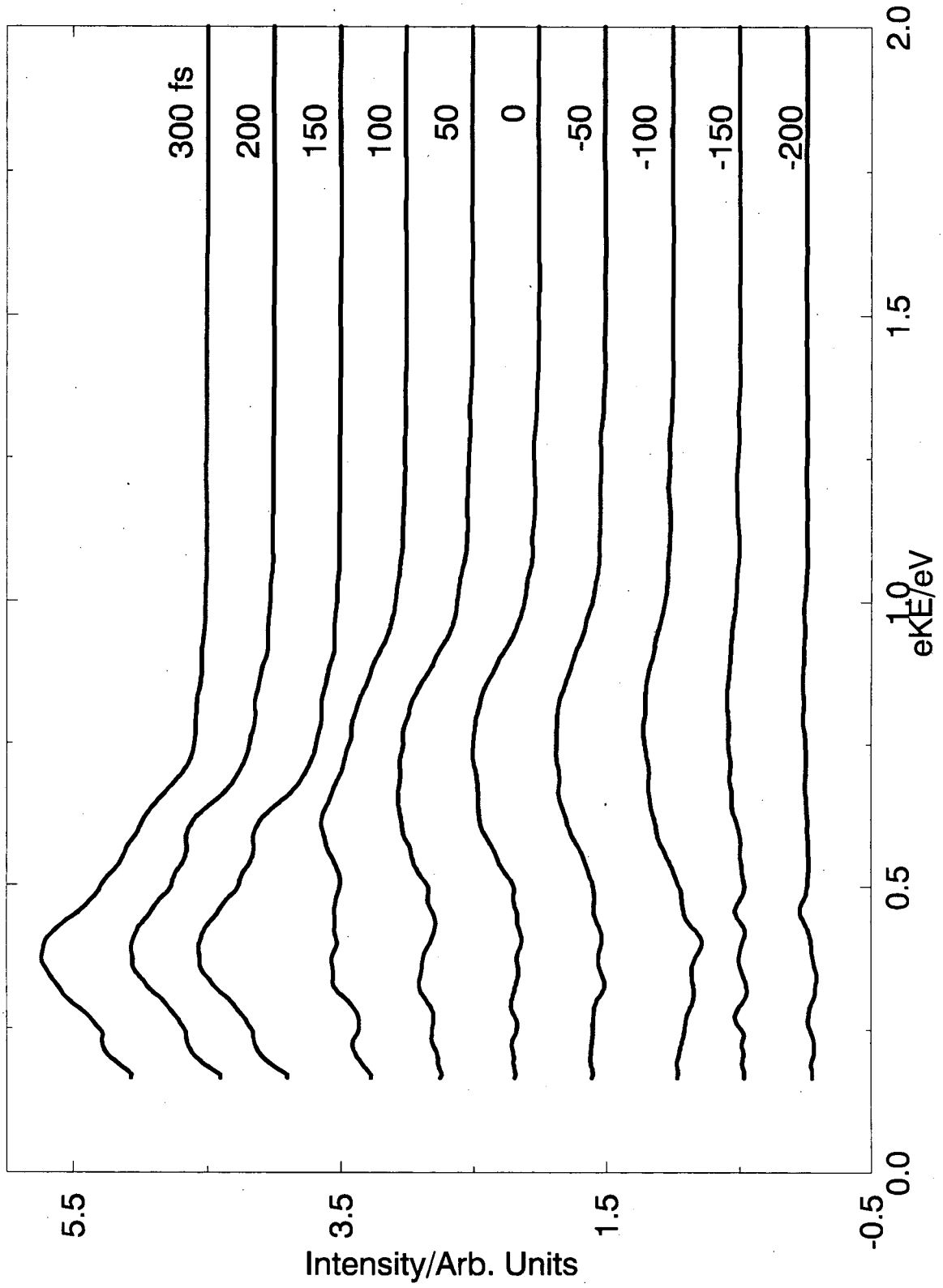

200000 fs
100000
50000
30000
20000
15000
12000
5X
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV

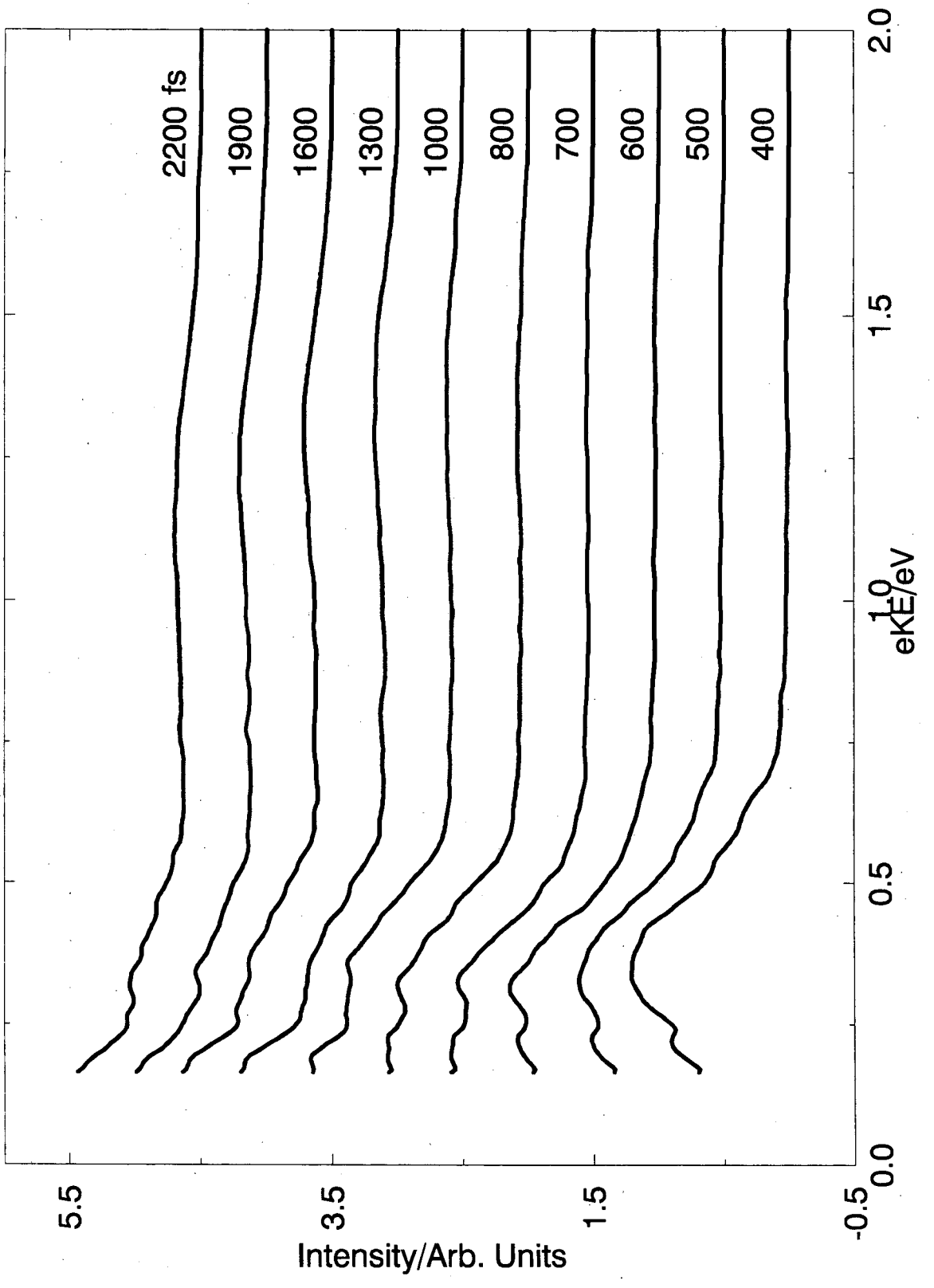
11. $I_2^-(CO_2)_{14}$

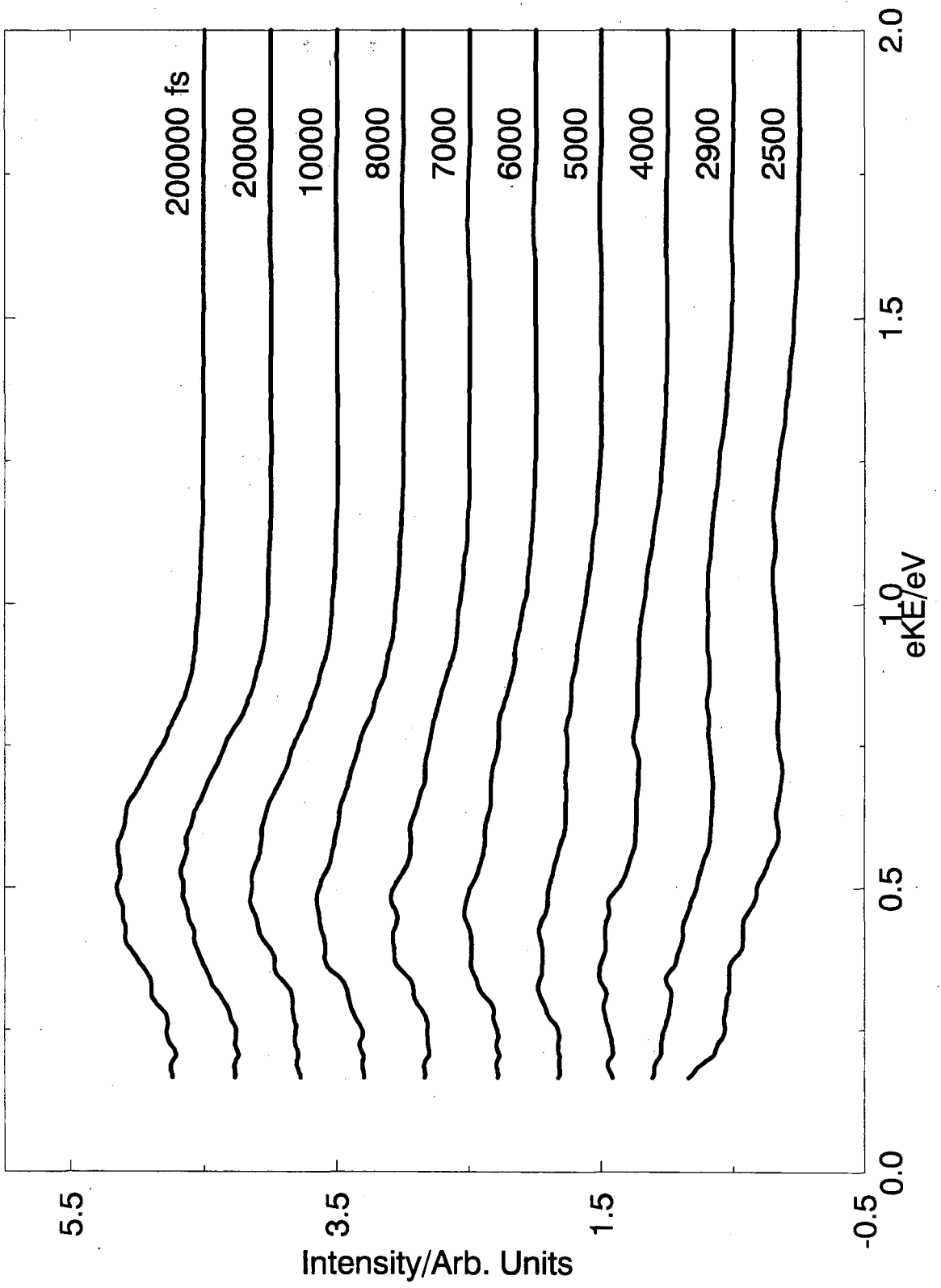

1300 fs
1000
880
730
630
530
430
330
230
30
3X
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV


200000 fs
100000
50000
20000
10000
7500
5000
2500
1500
3X
6.0
4.0
2.0
0.0
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV

12. $I_2^-(CO_2)_{16}$


300 fs
200
150
100
50
0
-50
-100
-150
-200
5.5
3.5
1.5
-0.5
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV


2200 fs
1900
1600
1300
1000
800
700
600
500
400
5.5
3.5
1.5
-0.5
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV


200000 fs
20000
10000
8000
7000
6000
5000
4000
2900
2500
5.5
3.5
1.5
-0.5
Intensity/Arb. Units
0.0
0.5
1.0
1.5
2.0
eKE/eV

Appendix 4. Publications from graduate work

1. B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, "Femtosecond photoelectron spectroscopy of $I_2^-(Ar)_n$ photodissociation dynamics (n = 6, 9, 12, 16, 20)," J. Chem. Phys., in preparation.
2. B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, "Femtosecond photoelectron spectroscopy of $I_2^-(CO_2)_n$ photodissociation dynamics (n = 4, 6, 9, 12, 14, 16)," J. Chem. Phys., in preparation.
3. M. T. Zanni, B. J. Greenblatt, A. V. Davis and D. M. Neumark, "Photodissociation studies of I_3^- using photoelectron spectroscopy," J. Chem. Phys., in preparation.
4. M. T. Zanni, V. S. Batista, B. J. Greenblatt, W. H. Miller, and D. M. Neumark, "Femtosecond photoelectron spectroscopy of the I_2^- anion: Characterization of the $\tilde{A}'\,^2\Pi_{g,1/2}$ excited state," J. Chem. Phys., **110**, 3748 (1999).
5. V. S. Batista, M. T. Zanni, B. J. Greenblatt, D. M. Neumark, and W. H. Miller, "Femtosecond photoelectron spectroscopy of the I_2^- anion: A semiclassical molecular dynamics simulation method," J. Chem. Phys., **110**, 3736 (1999).
6. M. T. Zanni, B. J. Greenblatt, and D. M. Neumark, "Solvent effects on the vibrational frequency of I_2^- in size-selected $I_2^-(Ar)_n$ and $I_2^-(CO_2)_n$ clusters," J. Chem. Phys. **109**, 9648 (1998).
7. M. T. Zanni, L. Lehr, B. J. Greenblatt, R. Weinkauf, and D. M. Neumark, "Dynamics of charge-transfer-to-solvent precursor states in $I^-(D_2O)_n$ clusters," Proceedings of the XIth Ultrafast Conference, Garmisch-Partenkirchen, Germany, in press (1998).

8. M. T. Zanni, B. J. Greenblatt, A. V. Davis, and D. M. Neumark, "Photodissociation dynamics of I_3^- using femtosecond photoelectron spectroscopy," Laser Techniques for State-Selected and State-to-State Chemistry IV, Proc. SPIE, **3271**, 196 (1998).

9. B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, "Time-resolved studies of dynamics in molecular and cluster anions," Faraday Discuss., **108**, 101 (1997).

10. M. T. Zanni, T. R. Taylor, B. J. Greenblatt, B. Soep, and D. M. Neumark, "Characterization of the I_2^- anion ground state using conventional and femtosecond photoelectron spectroscopy," J. Chem. Phys., **107**, 7613 (1997).

11. B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, "Photodissociation of $I_2^-(Ar)_n$ Clusters Studied with Anion Femtosecond Photoelectron Spectroscopy," Science, **276**, 1675 (1997).

12. B. J. Greenblatt, M. T. Zanni, and D. M. Neumark, "Photodissociation dynamics of the I_2^- anion using femtosecond photoelectron spectroscopy," Chem. Phys. Lett., **258**, 523 (1996).

ERNEST ORLANDO LAWRENCE BERKELEY NATIONAL LABORATORY
ONE CYCLOTRON ROAD | BERKELEY, CALIFORNIA 94720

Prepared for the U.S. Department of Energy under Contract No. DE-AC03-76SF00098