Title

EXPERIMENTAL METHODS AND SOFTWARE TOOLS FOR THE ANALYSIS OF ELECTROCHEMICAL SYSTEMS

Author

Matlosz, M.J.

Publication Date

1985-03-01

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Materials & Molecular Research Division

EXPERIMENTAL METHODS AND SOFTWARE TOOLS FOR THE ANALYSIS OF ELECTROCHEMICAL SYSTEMS

M.J. Matlosz
(Ph.D. Thesis)

March 1985

Prepared for the U.S. Department of Energy under Contract DE-AC03-76SF00098

DISCLAIMER

LBL-19375

Experimental Methods and Software Tools for the Analysis of Electrochemical Systems

Michael John Matlosz

Ph.D. Thesis

Lawrence Berkeley Laboratory
University of California
Berkeley, California 94720

Experimental Methods and Software Tools for the Analysis of Electrochemical Systems

Experimental Methods and Software Tools for the Analysis of Electrochemical Systems

Michael Matlosz

Abstract

This dissertation is concerned with the development of models of electrochemical systems and with the efficient and reliable analysis of those models. The presentation is divided into three parts, each a self-contained unit containing a summary, an introduction, and conclusions from the work.

Part 1 is an investigation of a porous electrode made of reticulated vitreous carbon for the removal of mercury from contaminated saltwater. Experiments with a bench-scale reactor show that the mercury concentration of contaminated saltwater solutions can be reduced by as much as a factor of five thousand in a single pass through the electrode. Results of the laboratory measurements are used to develop a correlation for the dependence of the mass-transfer coefficient on the flowrate of saltwater through reticulated vitreous carbon and to verify the validity of a mathematical model of the system.

Part 2 describes a functional style for writing computer programs to solve the one-dimensional boundary-value problems that arise frequently in the modeling of electrochemical systems. In addition, it presents a software tool, BandAid, that supports programming in this style. The programming methodology developed permits a static, self-documenting description of the differential equations to be placed directly in the program listing, thereby increasing the reliability of new programs and simplifying the maintenance and extension of existing ones. BandAid creates a finite-difference representation of the problem described in the program listing, and it uses a banded-matrix algorithm to solve the resulting set of algebraic equations.

Part 3 is a presentation of a fast, simple algorithm for simulating the transient current response of complex electrochemical reaction systems to arbitrary applied-potential waveforms. The simulation of cyclic voltammograms (current response to a triangular waveform) is emphasized. Development of a general-purpose software package is discussed, and simulated voltammograms for two electrochemical systems are compared to experiment.

John Newman

Table of Contents

Experimental Methods and Software Tools for the Analysis of Electrochemical Systems

Introduction

The Role of Research in the Design of Electrochemical Systems

Electrochemical systems are processes or devices that exploit interactions of electrical and chemical phenomena for practical gain. Examples include batteries, fuel cells, chlorine and aluminum manufacturing plants, metal refineries, and many other systems of economic importance. The fundamental study of electrochemical systems is intended to provide a rational basis for the design and construction of these processes. The goals of the research are to find general physical laws that represent the behavior of many diverse electrochemical systems and to develop methods for predicting the behavior of processes and devices based on these general laws.

Like all chemical process designs, electrochemical process designs necessarily require abstraction. Each part of a process must be idealized (modeled) in some way so that calculations about its size, shape, and operating conditions can be made. It is essential that the ways in which chemical processes are modeled provide reliable, accurate, and realistic representations of the characteristics that are important for scale-up, construction, and operation. Typical research projects, therefore, focus either on the verification of the validity of a particular model or on the determination of the best (most accurate) values for the parameters needed to use the model for scale-up.

According to a quantitative scientific approach, the validity of an abstraction can be verified by comparing calculations from a mathematical representation of the physical system to measurements from a laboratory-scale device. In this way, also, the best values for necessary parameters

can be obtained by surveying the agreement between measurements and calculations over a range of operating conditions. A complete research study, then, consists of two parts: development of a mathematical representation of a model system (mathematical model) and construction of a physical representation of that model in the laboratory (laboratory-scale device). Each of these two parts may be repeated many times before a successful match is found, and all of the work need not be completed by one individual or research group. Nevertheless, an investigation is not considered closed until both parts have been completed and acceptable agreement between the results has been obtained. Neither experiments nor calculations alone are sufficient for a complete investigation, since it is the quantitative agreement between model predictions and laboratory data that is necessary for the development of dependable process designs.

Outline of the Method of Investigation

At any point in a project, a particular abstract description of a physical system is being considered. If calculations are to be made, a mathematical representation of that abstract physical system is developed from the fundamental laws of physics and, in general, it involves construction of a set of equations that define the important process variables. If measurements are to be made, a laboratory-scale device is built to test some of the properties of the physical system and to see whether the abstract representation is valid on a small scale. To do this, the laboratory-scale device must exhibit those characteristics of the system that are to be checked, and, ideally, a model of the laboratory device should be based on some of the same physical laws as the large-scale process.

Measurements are made of the behavior of the laboratory device, and the measurements are compared to predictions calculated from the mathematical representation of the model. Since the process variables are normally defined by complicated sets of equations, it is often necessary to use a computer to calculate the predicted values of the process variables.

If the measurements agree with the calculations, further experiments are performed to determine the limits of applicability of the representation so that the use of the model for design calculations is safe and reliable. Values of the necessary design parameters can be refined, and the model can be combined with models of other parts of a process. In this way, the foundation for a complete process design can be built.

If the measurements do not agree with the calculated results, several possible causes must be investigated: the abstraction may be incorrect or incomplete; the physical laws may be invalid; the mathematical representation of the abstraction in terms of those physical laws may not be appropriate; or the laboratory device may exhibit characteristics other than those being tested. In general, any or all of these reasons must be considered, and it is a challenge to decipher the often conflicting messages received from an experimental investigation. These problems and challenges are particularly difficult in the study of electrochemical systems, where the interactions of electrical and chemical effects often make the analysis very complicated. In recognition of these difficulties, this dissertation is concerned with the development of models of electrochemical systems and with the efficient and reliable analysis of those models. It has two major themes:

– illustration of a quantitative scientific approach to the analysis of a particular system, a flow-through porous electrode, and

– development of useful software tools and techniques for the calculations required in the analysis of electrochemical systems.

The presentation is divided into three parts, each a self-contained unit containing a summary, an introduction, and conclusions from the work.

Part 1: Study of a Flow-Through Porous Electrode

In part 1 of this dissertation, an investigation of a laboratory device representing a flow-through porous electrode is discussed. The experiments and calculations are used to test an abstract representation of a flow-through porous electrode and to determine basic design

characteristics that any such reactor should exhibit. Several independent experiments were conducted so that the results of some of the experiments could be checked for consistency with the results from other experiments.

One major accomplishment was the development of a technique for the measurement of small concentrations of mercury in saltwater. The porous-electrode reactor is a very effective device, and the removal of mercury contamination in most experiments is nearly complete. In previous investigations of this process no mercury could be found in the effluent, and, therefore, no quantitative information for scale-up could be obtained. The use of a gold-film mercury analyzer in this study allowed the mercury in the effluent to be measured accurately so that the results of the measurements can be used to design other similar reactors.

A one-dimensional model of the flow-through porous electrode is used in the analysis of the results from the mercury experiments, and the mathematical description results in a set of coupled, nonlinear ordinary differential equations. Equations of this type arise frequently in the modeling of electrochemical systems, and the second part of this dissertation is concerned with the development of reliable software for solving these equations with a computer.

In addition to the flow-through porous electrode experiments, rotating disk studies were conducted to measure the diffusion coefficient of the mercuric chloride complex and to check the difference in reaction-rate constant for mercury deposition on solid glassy carbon and on the reticulated vitreous carbon (RVC) that formed the porous electrode. Linear sweep voltammetry was used for the rotating-disk studies, and the third part of this dissertation is concerned with the use of superposition integrals for the numerical simulation of such voltammetry experiments.

Parts 2 and 3: Development of General-Purpose Software Tools

Part 1 is a complete study, and it exhibits the characteristics of a full investigation described above. Parts 2 and 3 of the dissertation have a different flavor, since they are directed only at a single issue in the analysis process: the development of software for the reliable and

accurate determination of process variables from complex mathematical representations. Most important physical systems are complex, containing many interacting phenomena, and the determination of process variables requires the solution of sets of coupled equations. The aim of parts 2 and 3 is to develop software tools that can be used to solve reliably these complex equations.

Important characteristics of such a software package are that the equations being solved be easily stated for the computer and easily checked by the scientist doing the calculations. The work in part 2 is based on the belief that the style in which computer programs are written is an important consideration in the reliable development of large and complex software. A major accomplishment of the work in part 2, therefore, is a recommendation for a particular writing style appropriate for programs that solve differential equations, and the software tools developed in the dissertation support programming in this style.

Important though it is, however, reliability is not sufficient for computer programs, since a solution technique must also be efficient enough that large, complicated systems can be resolved on conventional computers in a short amount of time. The tools developed in parts 2 and 3 are both reliable and efficient, and it is hoped that they will prove useful to other researchers in the future.

Acknowledgements

It takes a long time to obtain a doctoral degree (27 years in my case), and no one does it alone. I have been fortunate to have found much support and encouragement throughout my entire life, and I am grateful to all those who have helped me along the way. My stay in Berkeley has been a happy one, in no small measure because of the many friends I have made, and I will miss them all very much.

I would like to express special thanks to Professor John Newman for providing a wonderful working environment these last five years. Professor Newman not only supported me with much needed advice but also showed a great deal of patience with a student who often marched to the

beat of a different drummer. Just as clear as his influence on the subject matter and technical content of this dissertation has been his influence on me and on my approach to solving problems. In the study of complex electrochemical systems, where attention to detail is of paramount importance, Professor Newman's direct, rigorous approach pays great dividends. I have come to appreciate the value of simply "doing it right," and I hope to be able to continue in that tradition in all my future work.

This work was supported by the United States Department of Energy under Contract No. DE-AC03-76SF00098 through the Director, Office of Energy Research, Office of Basic Energy Sciences, Chemical Sciences Division, and through the Assistant Secretary of Conservation and Renewable Energy, Office of Energy Storage Systems Research, Energy Storage Division.

Part 1. Experimental Investigation of a Porous Carbon Electrode for the Removal of Mercury from Contaminated Brine

1.1. Summary

A flow-through porous electrode, made of reticulated vitreous carbon (RVC), has been designed to remove mercury from contaminated brine solutions. Experiments with a bench-scale reactor show that the mercury concentration of contaminated brine solutions can be reduced by as much as a factor of five thousand during a single pass through the electrode. The process is mass-transfer limited, and the results of the experiments are used to develop a general correlation for the dependence of the mass-transfer coefficient on the flowrate of electrolyte through RVC. In addition, the effect of counterelectrode placement on the cell resistance is examined, and the experimental data are compared to predictions from a mathematical model of the system. The model agrees favorably with the experimental results, and the benefits of upstream counterelectrode placement, indicated by the model, are verified.

1.2. Introduction

The use of flow-through porous electrodes for the removal of heavy-metal ions from contaminated aqueous solutions has been discussed frequently in the electrochemical literature. The technique has been suggested for the recovery of several heavy-metal pollutants, including copper [1,2], silver [3,4], lead [5,6,7], and antimony [8], as well as mercury [9,10], gold, and cadmium. In all of these systems, the basic principle of separation is the same: the metal is removed by electrodeposition as the solution passes through a porous cathode of high surface area. Since, in many cases, this process allows the heavy metal to be recovered and sold as well as removed from solution, the technique may be economically attractive as an alternative to existing methods of waste

removal. Recent reviews of the subject and bibliographic information may be found in [9,11,12,13].

The study described here concerns the removal of mercury from contaminated brine (concentrated saltwater) and follows closely previous investigations of copper removal using a similar technique [1]. The results shown here represent the continuation of an ongoing study of mercury removal, and preliminary results from this laboratory, based on a slightly different reactor design, have been reported previously [10]. This earlier study on mercury indicated that the method is efficient and effective for the decontamination of brine solutions. In addition, the study showed that the mercury deposition system, due to its simplicity, is a good candidate for the general, theoretical study of flow-through porous electrodes. The experimental results of the present work are used to obtain quantitative information regarding the effectiveness of mercury removal under mass-transfer-limited conditions and to verify the applicability of the model of Trainham and Newman [14] to this system.

1.3. Electrochemical System

The principal electrochemical reactions that occur in the system are shown below:

Cathode: Mercury Reduction

$$HgCl_4^{-2} + 2e^- \rightarrow Hg(l) + 4Cl^-$$

Anode: Oxygen Evolution

$$2H_2O \rightarrow O_2 + 4H^+ + 4e^-.$$

In all of the experiments presented here, the catholyte was composed of a 4.3 M NaCl solution containing mercury concentrations of between 40 and 55 ppm. Mercury is highly soluble in such chloride solutions due to the complexing of the mercuric ion, and, in the range of concentrations and potentials of this study, $HgCl_4^{-2}$ ion is the predominant mercuric species. The solution

was slightly acidic (pH = 4), but hydrogen gas was not generated under typical conditions because the operating potential for the mercury deposition reaction is not sufficiently negative. At the anode, oxygen was evolved from an anolyte of the same salt concentration as the catholyte, but without the mercury. Although chlorine evolution at the anode is possible thermodynamically, none was observed under the conditions of this study.

Figure 1-1, a plot of the Nernst equation, shows the thermodynamic minimum mercury concentration attainable in a flow-through porous electrode as a function of the potential applied at the catholyte exit. An electrochemical method for the removal of mercury should be effective, since a very low equilibrium mercury concentration exists at a polarization of only a few hundred millivolts. (The abscissa ($V - \Phi_{ref}$) represents the potential of the working electrode with respect to a saturated calomel reference electrode (SCE).) In this study, an inlet concentration of mercury of 40 ppm corresponds to about -5 mV with respect to an SCE.

1.4. Electrode Configuration

Once a decision to attempt an electrochemical method of this type has been made, it becomes necessary to determine the arrangement of the electrodes that will constitute the reactor. Figure 1-2 shows two possible electrode configurations. Both are considered flow-through, a term indicating that the fluid flow and current flow are parallel. As a result, a one-dimensional model of a flow-through system should provide a suitable theoretical representation of the process. This may be contrasted with so-called flow-by configurations (not shown) in which the fluid flows in a direction perpendicular to the current. Mathematical models of flow-by systems must of necessity be two-dimensional.

Figure 1-2a shows upstream (before fluid inlet) placement of the counterelectrode, and figure 1-2b shows downstream placement. If the electrical conductivity of the solid packed bed is much higher than that of the electrolytic solution (which is the case here and is typical of practical porous-electrode systems), much better performance can be expected from upstream placement of

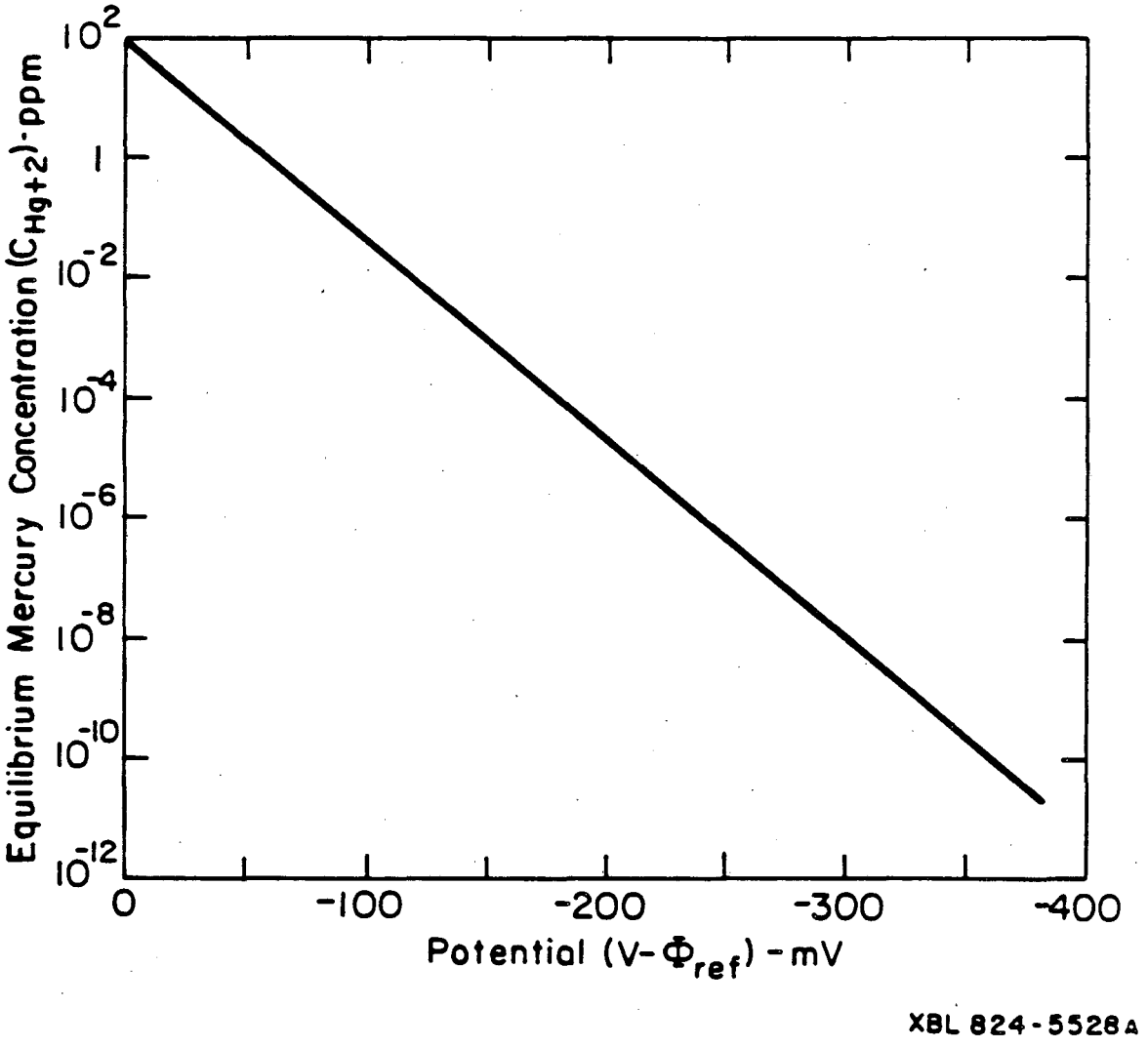


Figure 1-1. Equilibrium mercury concentration as a function of potential relative to a saturated calomel reference electrode.

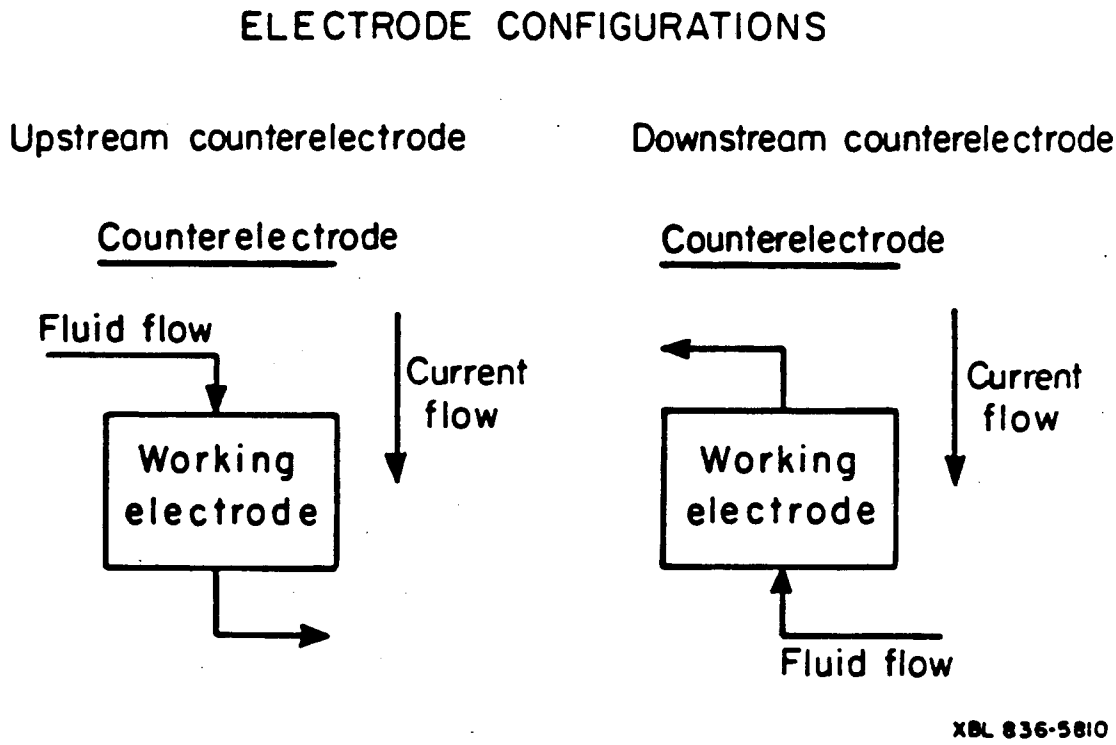


Figure 1-2. Counterelectrode configurations in a flow-through porous electrode system. (1-2a) Upstream counterelectrode placement. (1-2b) Downstream counterelectrode placement.

the counterelectrode than from downstream placement [11,15]. In the upstream-counterelectrode configuration, lower effluent concentrations are attainable, and the resistance loss (ohmic potential drop) is smaller than in the case of downstream placement. This smaller ohmic potential drop allows reactors with upstream placement to be operated at much higher flowrates without side reactions. Therefore, because of its practical importance, upstream-counterelectrode placement is emphasized in this study, and experiments involving downstream placement are presented only to demonstrate the effect of electrode placement on the ohmic potential drop.

1.5. Experimental Porous Electrode

Figure 1-3 shows a sketch of the experimental reactor. The cathode compartment is a plexiglass tube, two inches in diameter. A five-inch-long cylinder of RVC (E. R. G., Inc., Oakland, California) forms the working electrode (cathode), which is fitted atop a perforated current-collector plate. The anode compartment, a one-inch-diameter plexiglass tube, is separated from the cathode compartment by a Nafion membrane separator, and the counterelectrode (anode) is a Pt/Rh screen, spot-welded to a current-collector rod.

Isolation of the two electrode compartments permits independent control of the flows of anolyte and catholyte by two metering pumps (Fluid Metering, Inc.). Oscillations in the catholyte flowrate are removed by a flow damper placed after the catholyte pump, and the catholyte flowrate is measured by a rotameter (Gilmont Instruments, Inc.). Glass beads, placed above and below the carbon bed, distribute the fluid flow.

The anode and cathode current collectors are both made of tantalum (rather than stainless steel) to avoid corrosion due to the high chloride-ion concentration in solution. The flow system is constructed of Bev-A-Line chemical-resistant tubing connected by polypropylene tubing connectors. During the experiments, the feed solutions are sparged with nitrogen to remove oxygen that might be reduced at the cathode.

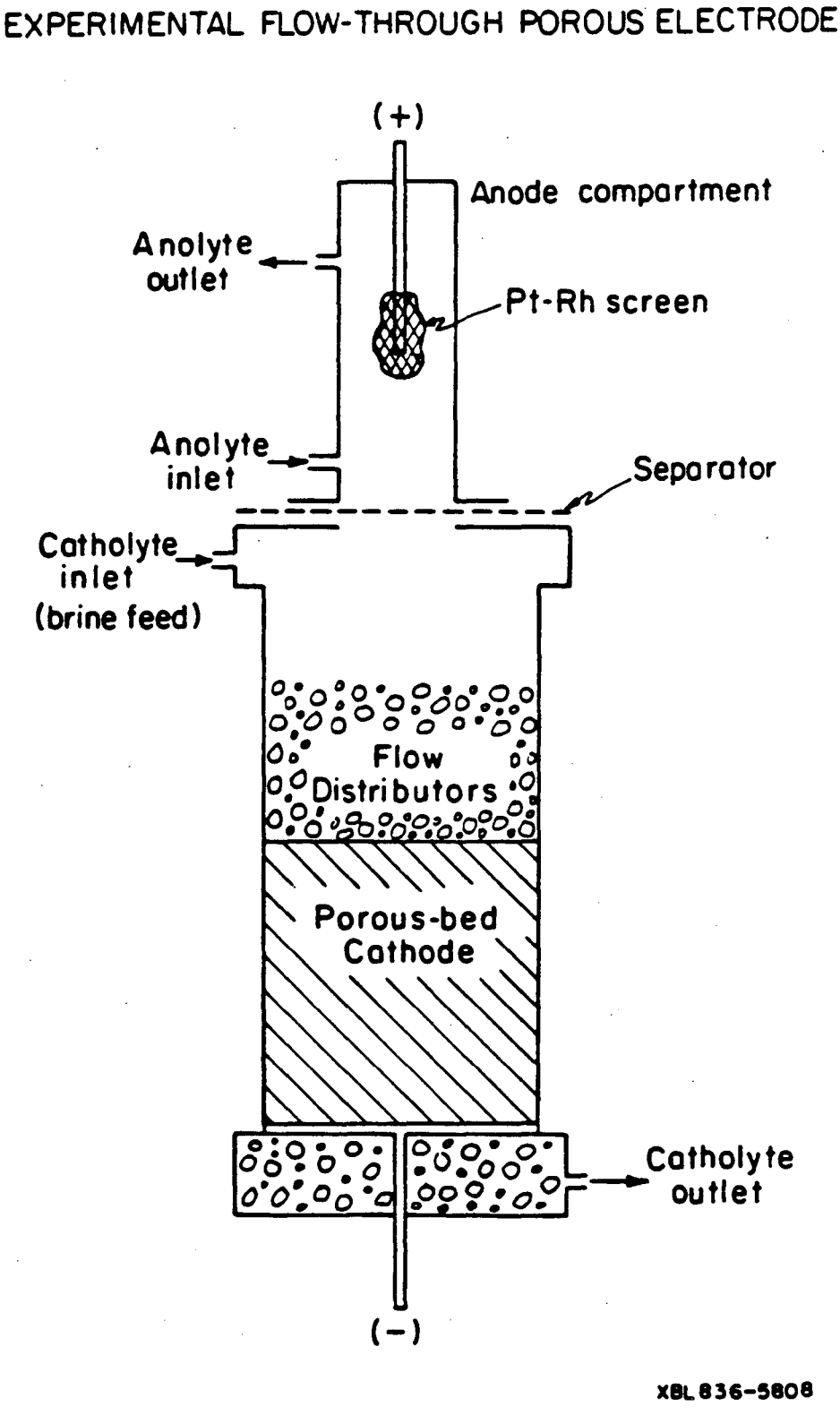


Figure 1-3. Sketch of the experimental flow-through porous electrode reactor.

Two reference electrodes (Corning Saturated Calomel Reference Electrodes) are placed in the system to monitor the solution potential. An upstream reference potential is measured from a capillary placed in the cell above the carbon bed, and a downstream reference potential is measured at the catholyte outlet. All experimental polarization curves are obtained under potentiostatic control, where the potential of the working electrode (cathode) with respect to the saturated calomel reference electrode in the exit stream is regulated by an AIS Model V-2LR-D Potentiostat.

Photographs of the reactor, the carbon electrode, and the flow system are shown in [6], which describes additional experiments, with the same reactor, for lead removal from contaminated sulfuric acid solutions.

1.6. Electrode Operation and Concentration Measurement

Figure 1-4 illustrates the general operation of the porous electrode. To remove mercury, the working electrode is polarized cathodically, and current is drawn from the current collector. Mercury is deposited onto the surface of the electrode as the contaminated solution passes through it, and oxygen is evolved at the counterelectrode. Once a solution has been purified, the mercury can be recovered, and the electrode can be regenerated, by reversing the polarization of the cell. Thus, in the regeneration operation, the working electrode is polarized anodically, mercury on the electrode surface is dissolved, and hydrogen is evolved on the counterelectrode.

Since the mercury can be stripped very quickly and easily from the electrode during the regeneration step, this process provides a convenient method for concentrating dilute mercuric chloride solutions. Initially, the dilute solution can be purified (and the mercury stored on the electrode surface). Then, by decreasing the flowrate of brine through the electrode during the regeneration step, a small volume of concentrated mercuric chloride solution can be obtained, from which product mercury may be recovered by conventional electrowinning techniques.

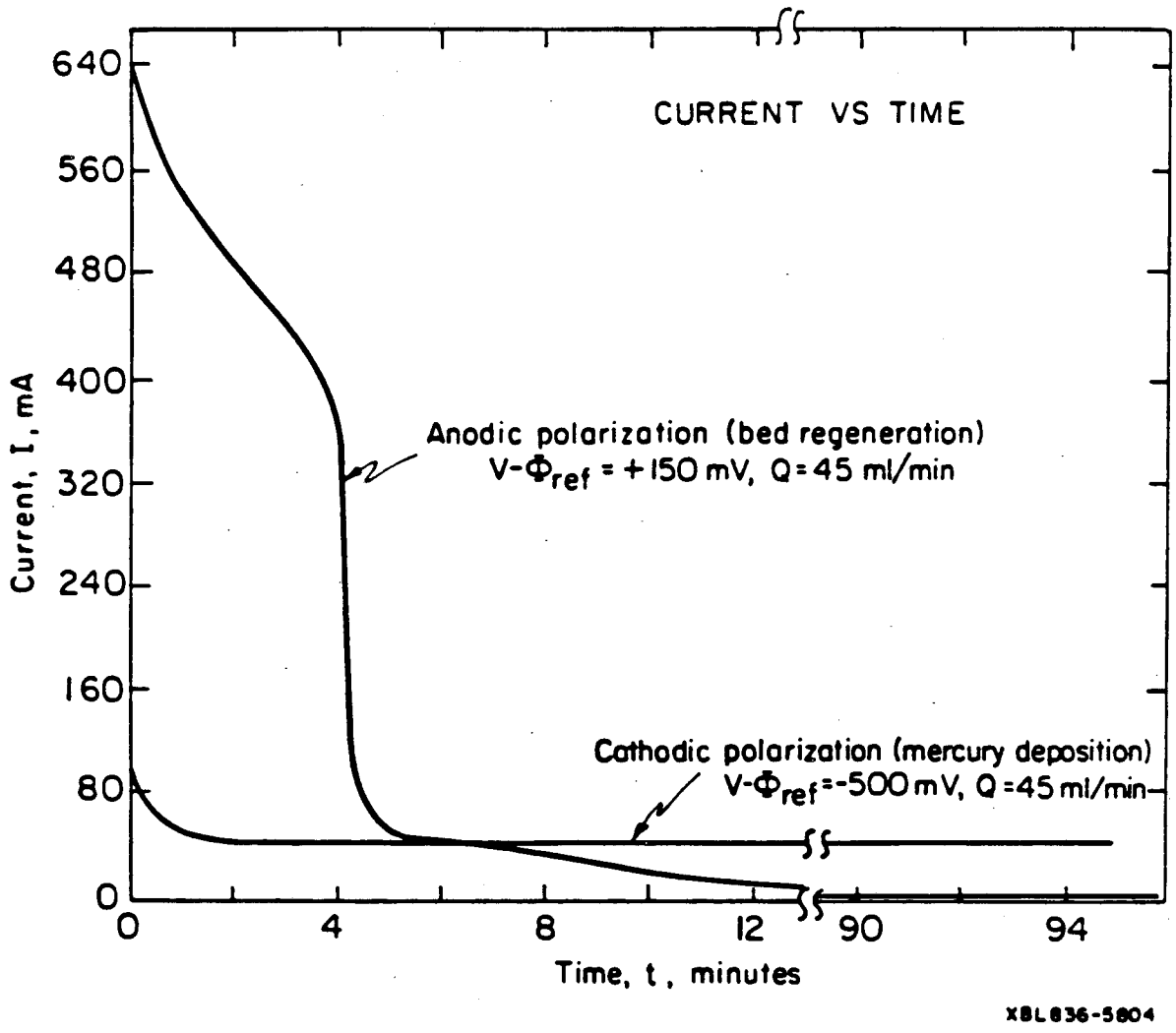


Figure 1-4. Current to the porous electrode as a function of time. (Current is plotted as the absolute value.)

This two-step procedure has many practical applications, and, although this study focuses principally on the mercury-deposition (purification) step, bed regeneration was in fact used in the laboratory. By recovering the metal as described above, mercury deposited onto the RVC in one experiment could be reused in another, the pore characteristics of the RVC were preserved, and pore plugging was avoided.

Accurate analysis of the concentration of mercury in the effluent solutions was essential if the experimental results were to be useful for process scale-up. Unfortunately, because of the extremely low concentrations of mercury (low ppb) as well as the high concentration of chloride ion (4.3 M), conventional measurement techniques (such as atomic absorption spectrophotometry) were not adequate. As a result, it was necessary to employ a more sensitive method for the measurements -- a gold-film mercury analyzer.

The gold-film analyzer measures the change in the electronic resistance of a piece of gold foil upon contact with an air stream containing mercury vapor, and the instrument is sensitive to as little as one nanogram of elemental mercury. In addition, since the resistance change is due to the amalgamation of the mercury with the gold (a process unique to mercury), the measurement is specific and free of interferences.

For this study, mercury concentrations in all of the liquid samples were measured with a Jerome Instruments, Inc., Model 301 Gold Film Mercury Analyzer according to the procedure outlined in [16]. First, mercuric ions in the liquid samples were chemically reduced (with $SnCl_2$) to elemental mercury. Next, the reduced solution was sparged with a pure air stream in order to liberate mercury vapor, and the air stream was then passed to the analyzer, where the total mass of mercury was measured. Finally, the mercury concentration was determined from the mass of mercury vapor detected divided by the volume of the original (liquid) sample. With this technique, mercury concentrations from one part per thousand down to five hundred parts per trillion could be measured to an accuracy of about ten percent.

1.7. Experimental Results From the RVC Reactor

1.7.1. Steady-State Polarization Behavior

Figures 1-5 and 1-6 show the steady-state polarization behavior for mercury deposition in the RVC reactor for both the upstream- and downstream-counterelectrode configurations. The points are the polarization measurements, and the numbers below the points show the experimentally determined mercury concentration (in ppm) at the catholyte exit. The curves on the figures represent predictions from the mathematical model.† Flat limiting-current plateaus were obtained with only moderate polarization, indicating that the major limitation to mercury removal is the rate of mass transfer of the mercuric-ion complex to the surface of the cathode pore. In a typical experiment, about 3000 to 5000 cm^3 of solution were passed before steady state was achieved. Thus, in this case, for a flowrate of 30 cm^3/min, each point on the polarization curve required 100 minutes of stable operation of the reactor. Sparging of the catholyte feed with nitrogen reduced the oxygen content of the incoming brine, and faradaic current efficiencies of about 90 percent were achieved routinely. At high polarization (above -500 mV), the increase in current is due to the production of dissolved hydrogen gas. Hydrogen bubbles, however, were not produced, since the amount of gas generated was well below the solubility limit. Table 1-1 indicates pertinent physical property data†† and operating conditions for the measurements shown in figures 1-5 and 1-6.

1.7.2. Effect of Counterelectrode Placement

Figure 1-7 demonstrates the effect of counterelectrode placement on the resistance to current flow within the porous electrode. The abscissa is the applied potential (in mV) between the work-

† A discussion of the fitting parameters used for the model predictions is presented later in section 1.10.

†† Values of U_S^θ, U_{re}, ρ_o, μ, ν, κ_o, and $P_{H_2,f}$ are taken from [17]; a, ϵ, and σ from [18]; U_R^θ from [9]. Other values were measured in the laboratory.

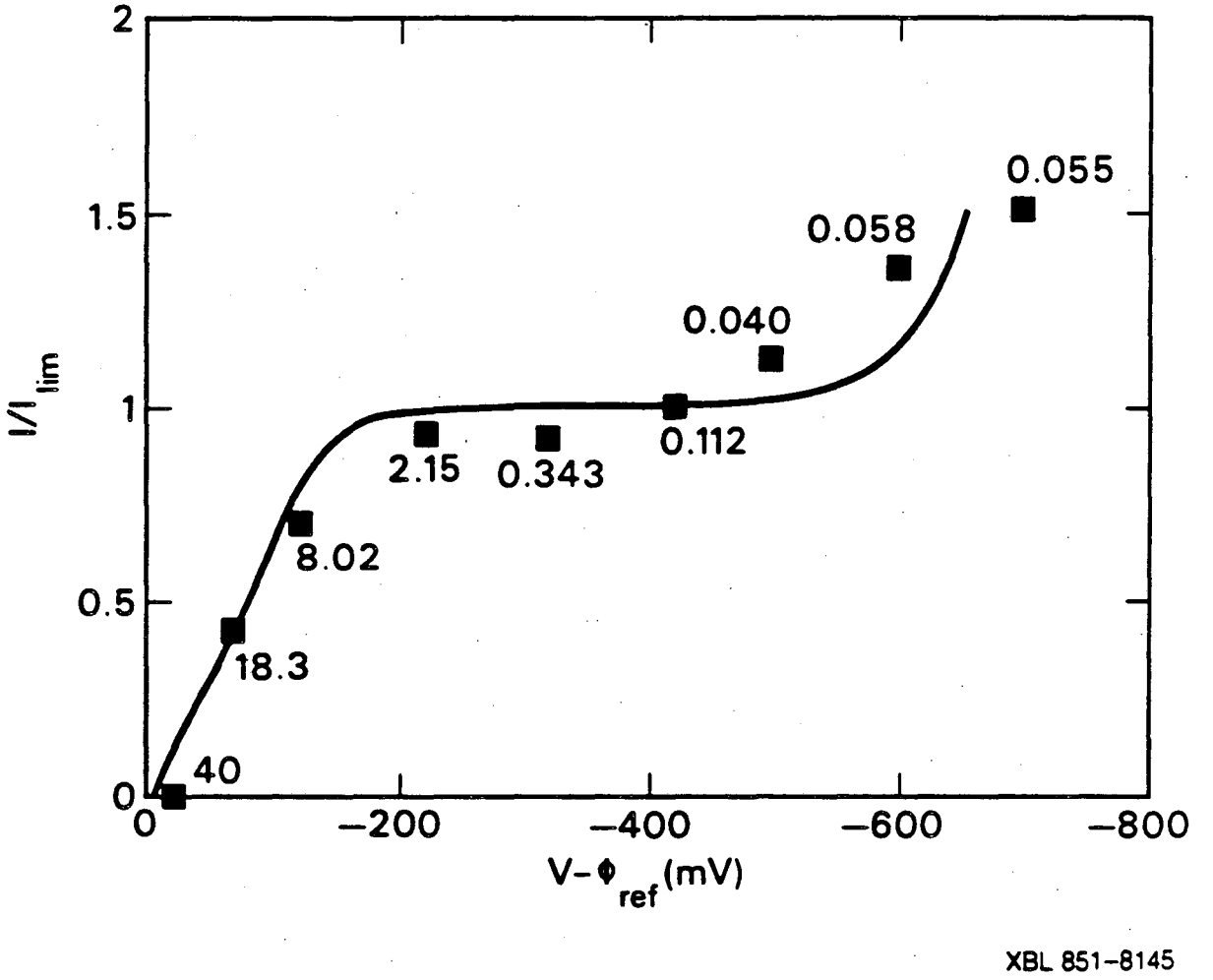


Figure 1-5. Polarization curve for the reactor with upstream counterelectrode placement. Q = 30 cm^3/min , L = 12.7 cm, Feed concentration = 40 ppm Hg (c_{Rf} = 2.273 $\times 10^{-7}$ mol $/cm^3$). The points are experimental measurements, and the curve represents the model. Numbers below the points show experimentally measured effluent concentrations.

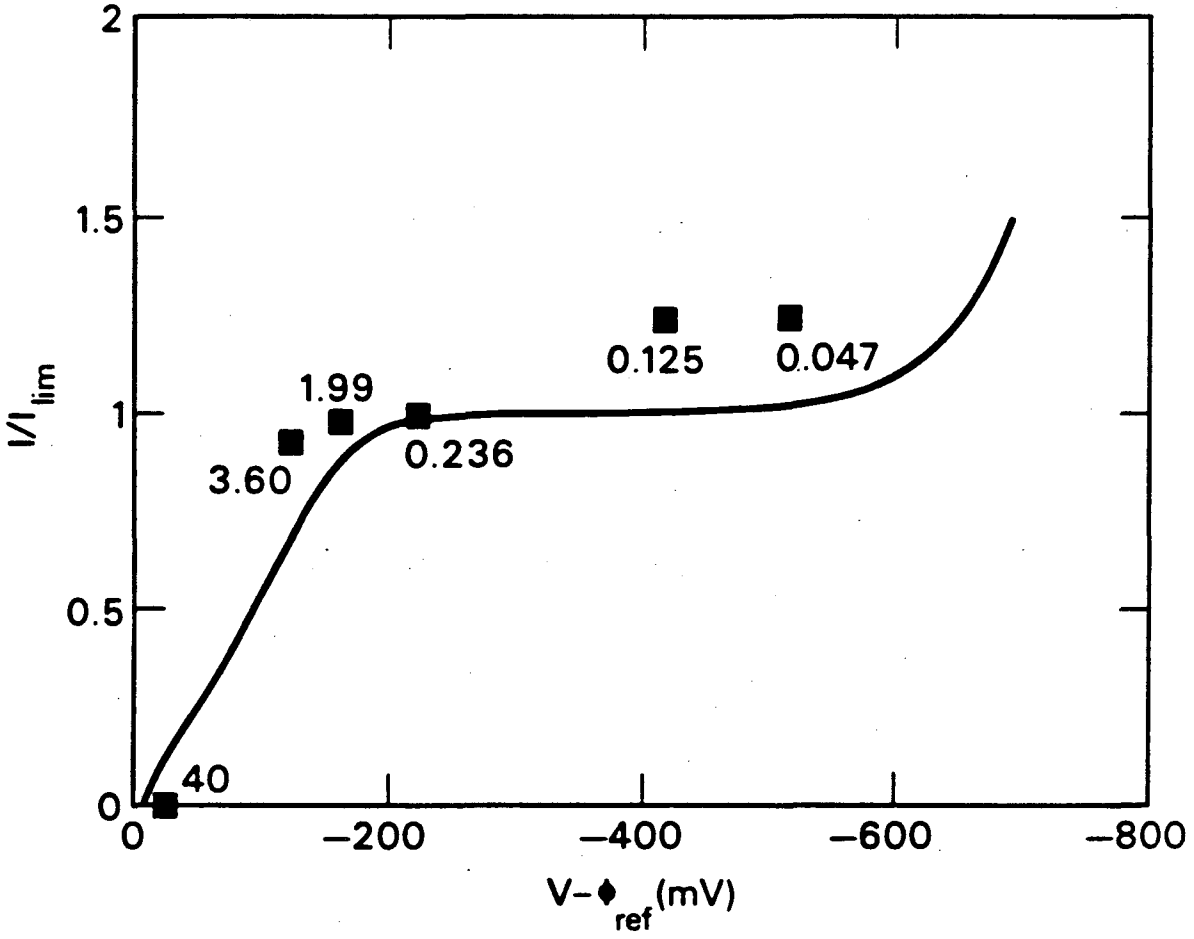


Figure 1-6. Polarization curve for the reactor with downstream counterelectrode placement. Q = 30 cm^3/min, L = 12.7 cm, Feed concentration = 40 ppm Hg (c_{Rf} = 2.273 $\times 10^{-7} mol/cm^3$). The points are experimental measurements, and the curve represents the model. Numbers below the points show experimentally measured effluent concentrations.

Table 1-1. Physical Property Data and Operating Conditions

a =	66 cm^2/cm^3	U_R^θ =	0.4138 V ($HgCl_4^{-2}$ /Clsup -/Hg)
D_o =	1.0×10^{-5} cm^2/s	U_S^θ =	0.000 V (H^+/H_2)
ϵ =	0.97	U_{re} =	0.2415 V (Calomel, Sat'd KCl)
ρ_o =	1.14×10^{-3} kg /cm^3	μ =	1.52×10^{-2} g /cm -s
ν =	1.333×10^{-2} cm^2/s	κ_o =	0.199 mho/cm
σ =	1.73 mho/cm	Sc =	1333
c_{RJ} =	2.273×10^{-7} mol /cm^3	$c_{Cl^-,J}$ =	4.3×10^{-3} mol /cm^3
L =	12.7 cm	pH =	4.0
$P_{H_2,J}$ =	5×10^{-7} atm	S =	20.26 cm2
T =	298.15 K	v =	0.0255 cm/s
$\bar{k}_m$ =	2.102×10^{-4}cm /s	Re =	0.02811
Pe =	38.64	Sh =	0.3089

ing (porous) electrode and a saturated calomel reference electrode at the catholyte exit. The ordinate represents the effective area-specific resistance,

$$\frac{\Phi_{upstream} - \Phi_{downstream}}{i},$$

and has units of ohm-cm^2. As in figures 1-5 and 1-6, the points are the experimental measurements, and the curves represent model predictions. (During the experiments, the tip of the capillary from the upstream reference electrode was located a distance of 1 cm above the porous electrode, within the flow distributors, resulting in an uncompensated resistance of 30 ohm-cm^2.† This value is added to the resistance calculated from the model.)

Although the removal effectiveness is approximately the same for both cases (see figures 1-5 and 1-6), it can be seen that downstream placement causes a considerable increase in ohmic potential drop. This difference can be understood by considering the concentration distribution through the electrode under mass-transfer-limited conditions. In the absence of axial diffusion and

† The glass beads represent a bed of uniformly packed spheres, $\epsilon = 0.3$. (The effective conductivity, κ, is calculated as shown in table 1-3.)

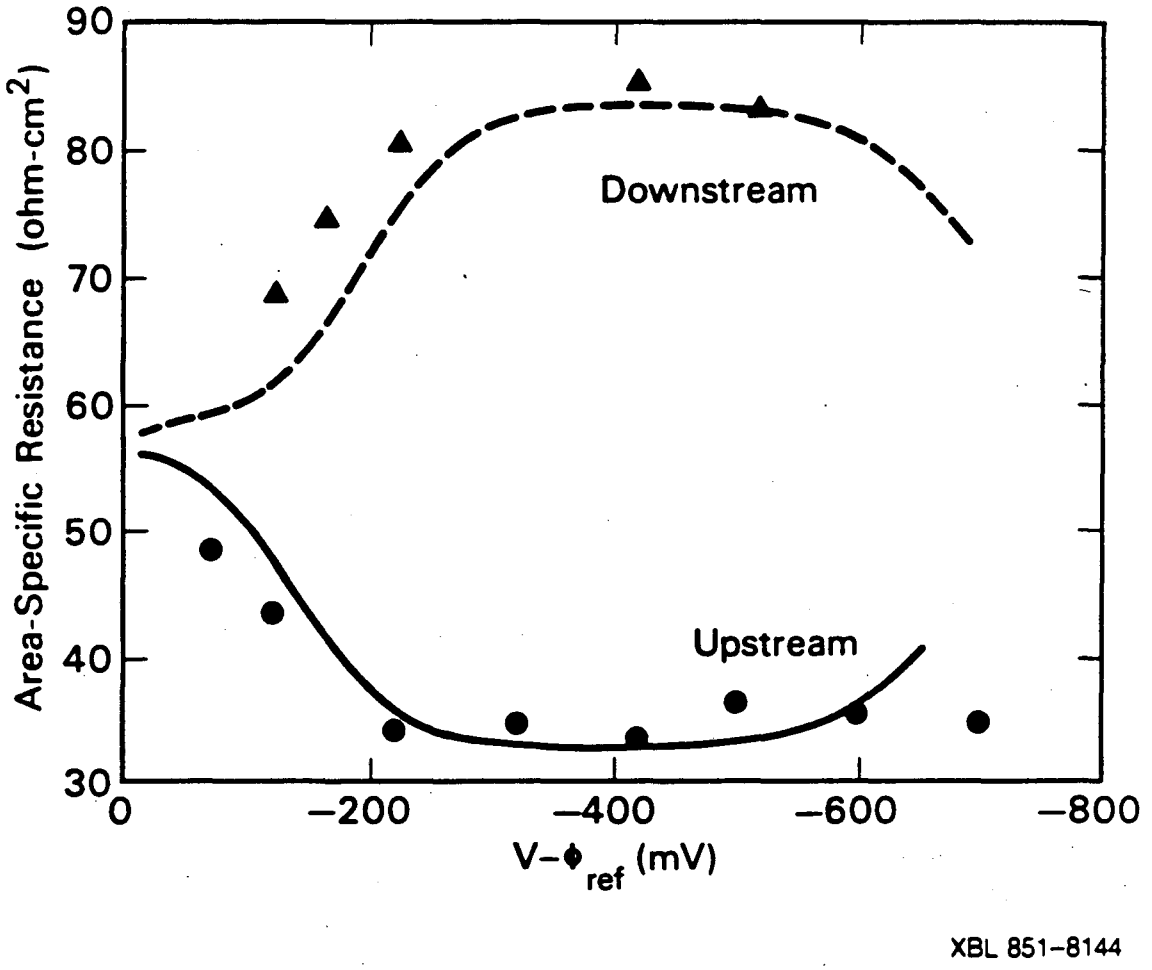


Figure 1-7. Effect of counterelectrode placement on the area-specific solution resistance within the porous electrode. (Operating conditions are identical to those in figures 1-5 and 1-6.) The points are experimental measurements, and the curves represent the model.

dispersion effects, the concentration drops exponentially with length x through the electrode, as represented by the following relation [1]:

$$\frac{c_R(x)}{c_{Rf}} = \exp\left(-\frac{ak_m x}{v}\right), \tag{1-1}$$

where k_m represents a local mass-transfer coefficient, assumed to be constant throughout the length of the reactor. Examination of this relationship reveals that the majority of the mercuric-ion complex has been removed within a short distance, of order v/ak_m, of the entrance to the reactor. Thus, the bulk of the charge transfer occurs in a region generally much shorter than the total length of the reactor, L. Although the additional length is necessary when high removal effectiveness is desired, the reduction of trace amounts of mercury further along the reactor contributes only a small fraction to the total charge transfer.

Figure 1-8 illustrates the effect of counterelectrode placement on the overall resistance by showing the "effective" current path through the electrolyte for the two cases. In the case of upstream placement, the current must travel only a distance equivalent to approximately v/ak_m, whereas in the case of downstream placement, the current path is approximately equivalent to the length of the reactor, L. If high removal effectiveness is desired, then L will be much greater than v/ak_m, and, consequently, the resistance will be much higher for downstream placement than for upstream placement of the counterelectrode. (Only the current path through the electrolyte is considered here, since the conductivity of the carbon bed is much higher than that of the electrolyte. If the conductivity of the electrode matrix is of the same order as that of the electrolyte, then the placement of the cathode current collector is also important. For a discussion of this effect, see [15].)

In general, the additional cell resistance in the case of downstream placement causes difficulties in the operation of the reactor, since the possibility of a side reaction is increased considerably. Furthermore, this increased likelihood of side reaction has a direct effect upon the reactor design. In particular, for high removal efficiency, the current density is directly proportional

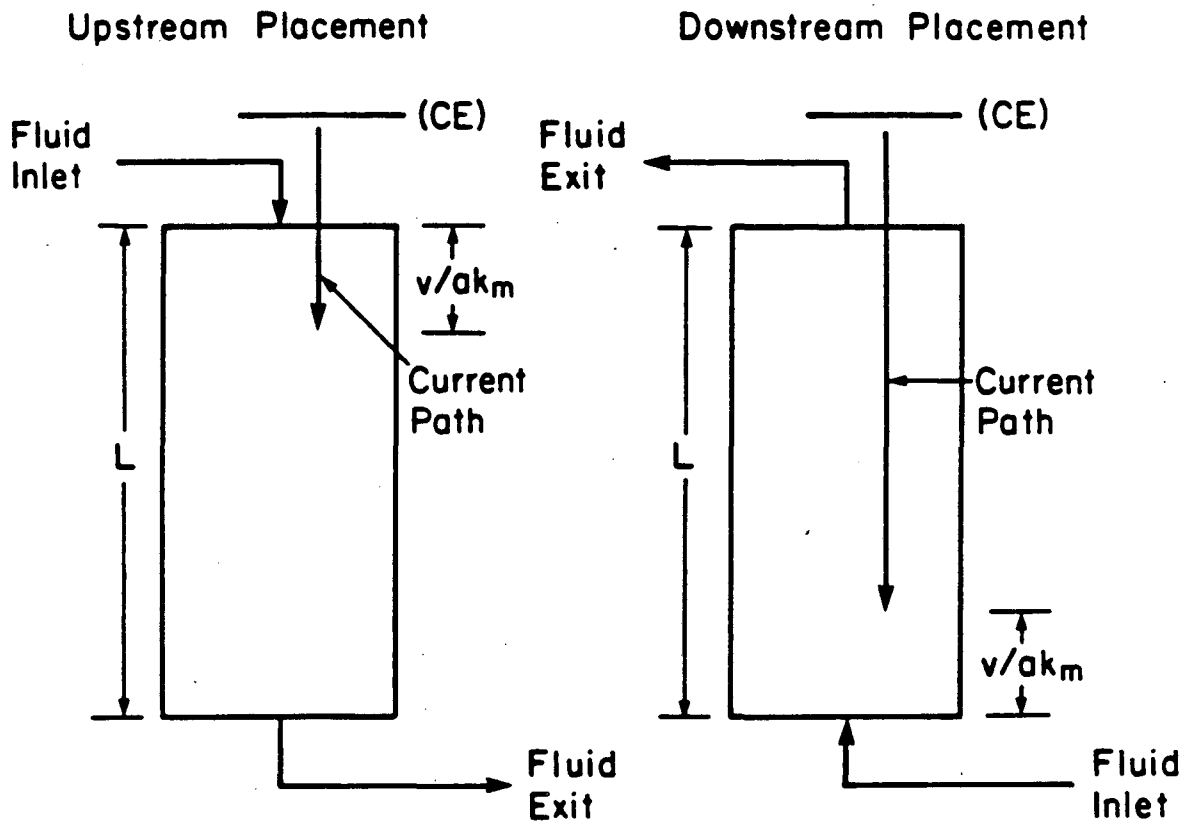


Figure 1-8. Sketch of the effect of counterelectrode placement on the effective current path through the electrolyte.

to the flowrate of catholyte, and the ohmic potential drop is directly proportional to the current density. Thus, if the ohmic potential drop, $\Phi_{upstream} - \Phi_{downstream}$, must be kept below some critical value in order to avoid side reactions, the maximum permissible flowrate is higher with upstream counterelectrode placement than with downstream placement. In short, the higher resistance in the downstream counterelectrode configuration limits the throughput of the reactor [1,11].

Under the conditions of this study, mercury removal is not affected by counterelectrode placement (see figures 1-5 and 1-6), since a very high overpotential is required for the production of hydrogen and the flowrates are not near their limiting values. Nevertheless, the resistance measurements shown in figure 1-7 indicate that, in this reactor, the maximum permissible flowrate for the upstream-counterelectrode configuration is significantly larger than that for the downstream configuration. The remainder of the results presented in this paper will be restricted to cases of upstream counterelectrode placement in the reactor.

1.7.3. Effect of Catholyte Flowrate

Figure 1-9 illustrates the effect of catholyte flowrate on the limiting current. As before, both the experimental data and the model predictions are shown. Since the removal of mercury is very nearly complete, the limiting current increases proportionately with catholyte flowrate.

The effect of catholyte flowrate on the effluent concentration (at limiting current[†]) is summarized in figure 1-10 and table 1-2 for seven experimental runs. Increased convection in the cathode pores at higher flowrates reduces the mass-transfer resistance. Nevertheless, the effect of the associated decrease in residence time is greater, and, therefore, the effluent contains more mercury at the higher flowrates.

The effectiveness of this method of removing mercury from contaminated solutions is clearly

[†] All limiting-current data were obtained at a polarization, $(V-\Phi_{ref})$, of -500 mV.

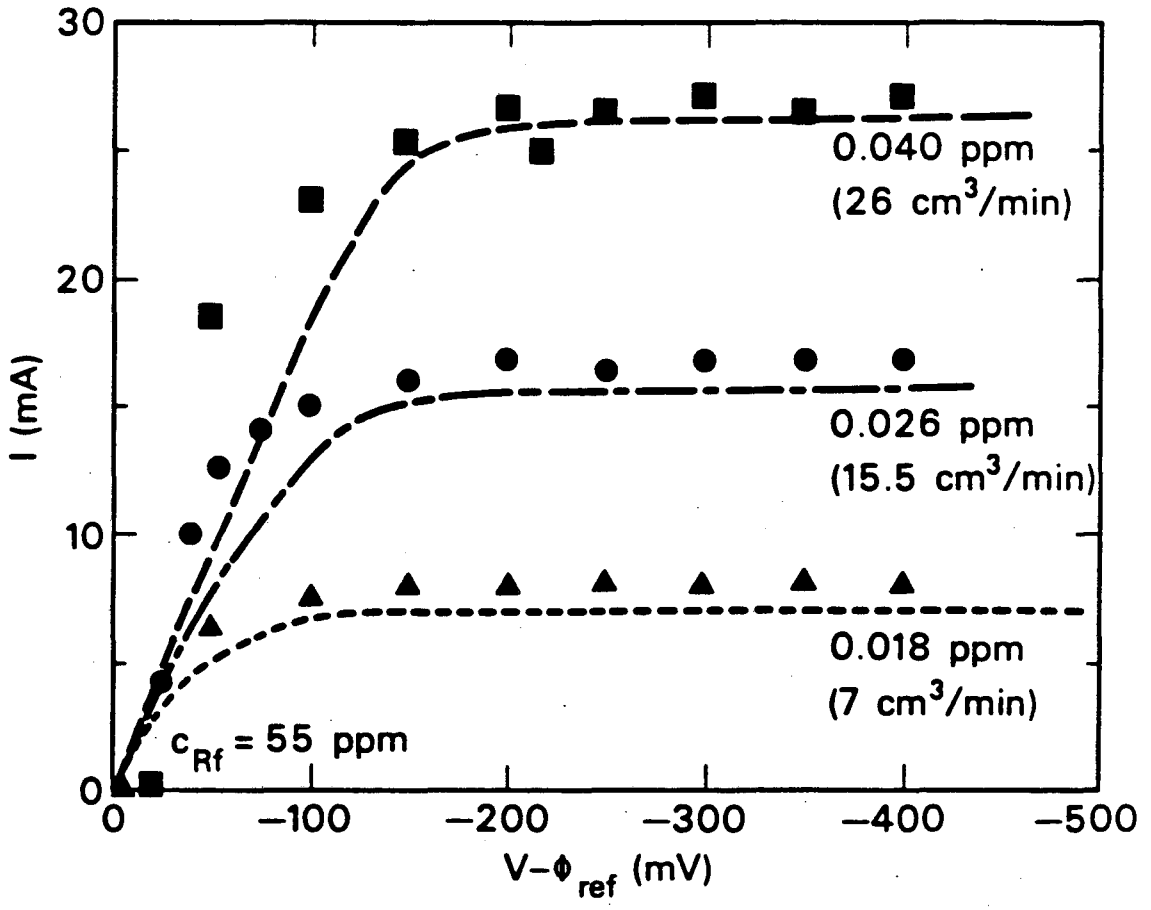


Figure 1-9. Polarization curves as a function of catholyte flowrate. L = 12.7 cm, feed concentration = 55 ppm Hg. The points are experimental measurements, and the curves represent the model.

Table 1-2. Effect of Flowrate on Effluent Concentration at Limiting Current

Q (cm^3/min)	v (cm/s)	c_{Rf} (ppm Hg)	$c_R(L)$ (ppm Hg)	$\bar{k}_m$ (cm/s)
98	0.0806	26	0.606	3.615×10^{-4}
65	0.0534	65	0.262	3.513×10^{-4}
45	0.0370	110	0.168	2.862×10^{-4}
31	0.0255	59	0.0588	2.102×10^{-4}
22	0.0181	42	0.0303	1.562×10^{-4}
15	0.0123	56	0.0262	1.125×10^{-4}
10	0.00822	55	0.0180	7.870×10^{-5}

demonstrated by these results. At the lowest flowrate examined (10 cm^3/min), a decrease of a factor of 5000 in the mercury concentration was achieved, indicating that this method is a feasible alternative to existing chemical methods of mercury removal.

1.7.4. Definition of the Mean Mass-Transfer Coefficient

Equation (1) can be used to develop the following general definition:

$$\bar{k}_m = -\frac{v}{aL}\ln\left(\frac{c_R(L)}{c_{Rf}}\right), \tag{1-2}$$

where the mean mass-transfer coefficient, $\bar{k}_m$ (unlike the local coefficient, k_m) may contain the effects of axial diffusion and dispersion. $\bar{k}_m$ is more convenient than k_m for tabulation, since its use does not require an independent value of the dispersion coefficient [11,19]. The mean mass-transfer coefficient, $\bar{k}_m$, in general, depends on the diffusion coefficient (D_e), the solution velocity (v), the viscosity (μ), and the electrode geometry (pore structure). Therefore, a generalized dimensionless correlation of the effect of velocity on the mass-transfer coefficient in RVC can be developed from the results of these experiments, provided that independent measurements of the surface area per unit volume of reactor (a) and the diffusion coefficient (D_e) are available. Reticulated vitreous carbon has a uniform pore structure, and the value of surface area per unit volume is available [18]. The value of the diffusion coefficient, however, has not been measured, and it was therefore decided to determine its value in an independent experiment.

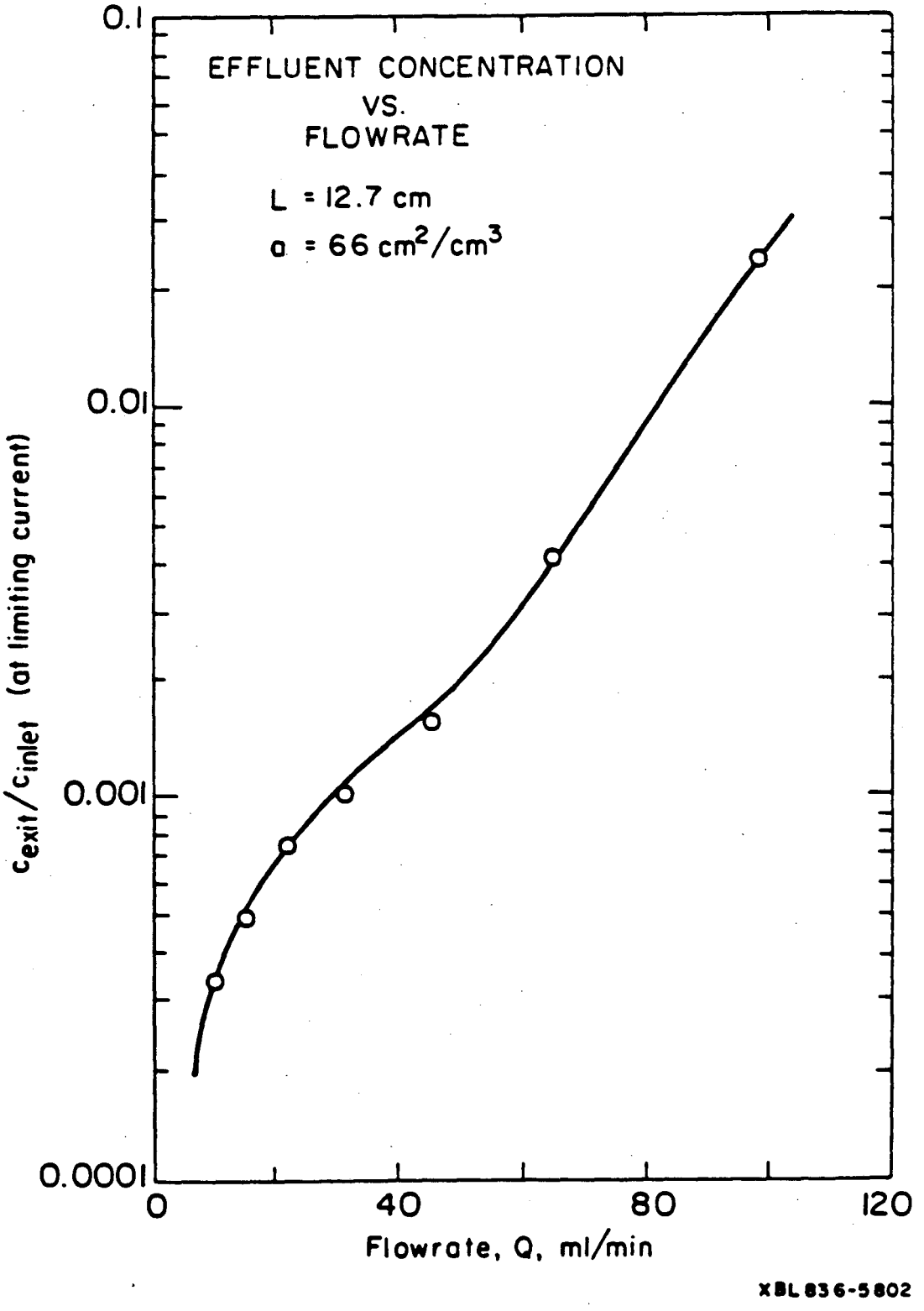


Figure 1-10. Effect of flowrate on the effluent concentration at limiting current.

1.8. Measurement of the Diffusion Coefficient

The diffusion coefficient of the mercuric chloride complex was obtained from the limiting current to a rotating disk electrode. For a single-electrode reaction on the surface of a disk, an integral-average diffusion coefficient of the electro-active species in solution can be determined from the Levich relation shown below [20]:

$$I_{\text{lim}} = 0.620nFAD_o^{2/3}\Omega^{1/2}\nu^{1/6}c_b . \tag{1-3}$$

A plot of the mass-transfer-limited current, I_{lim}, versus the square root of the angular velocity, $\Omega^{1/2}$, is a straight line, and the diffusion coefficient may be obtained directly from the slope.

Figure 1-11 shows a sketch of the apparatus used for the rotating-disk experiments. A Pine Instruments potentiostat regulated the potential of the working, glassy-carbon electrode (A = 0.442 cm^2) with respect to a saturated calomel reference electrode in the side arm. Figure 1-12 shows the resulting polarization curves (for a sweep rate of 5 mV/sec). Flat, stable limiting-current plateaus were obtained, and a plot of the limiting current as a function of the square root of rotation speed[†] is shown in figure 1-13 for a concentration of 150 ppm Hg in 3.8 M NaCl. From the slope of the plot, a diffusion coefficient of $(1.0 \pm 0.2)\times10^{-5}$ cm^2/s was determined.

1.9. Generalized Correlation for the Mass-Transfer Coefficient

If the velocity and mass-transfer coefficient are non-dimensionalized with respect to a characteristic length (a^{-1}) and diffusion coefficient (D_o), then a correlation of the behavior of the mass-transfer coefficient as a function of velocity should depend only on geometric factors (*i. e.*, pore structure of the RVC). Such a plot can now be obtained from the experimental results of the porous-electrode experiments, and it is shown in figure 1-14. The dimensionless mass-transfer coefficient, or Sherwood number (Sh), defined as

[†] Notice that, in figure 1-13, Ω is expressed in revolutions per minute (rpm).

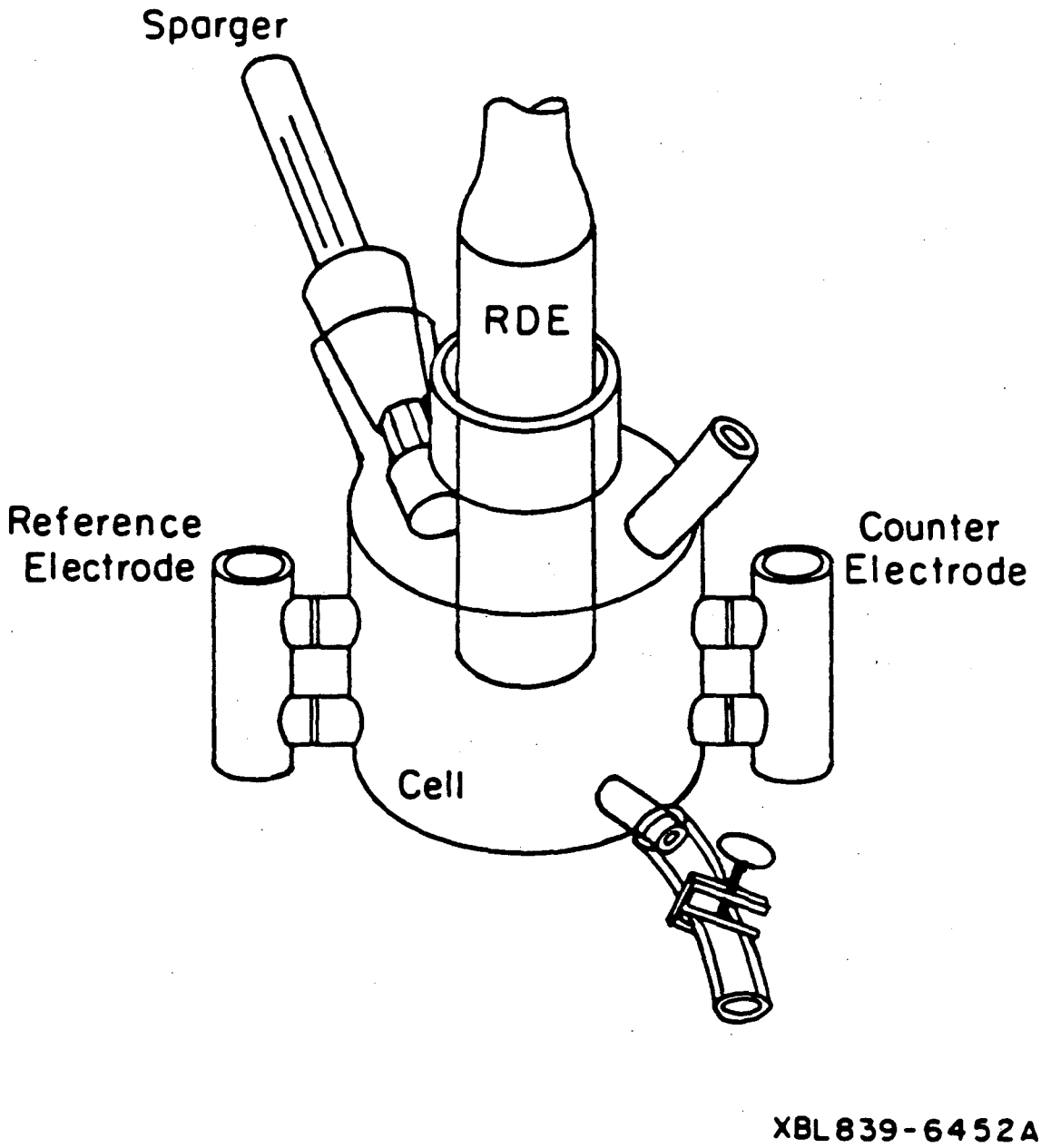
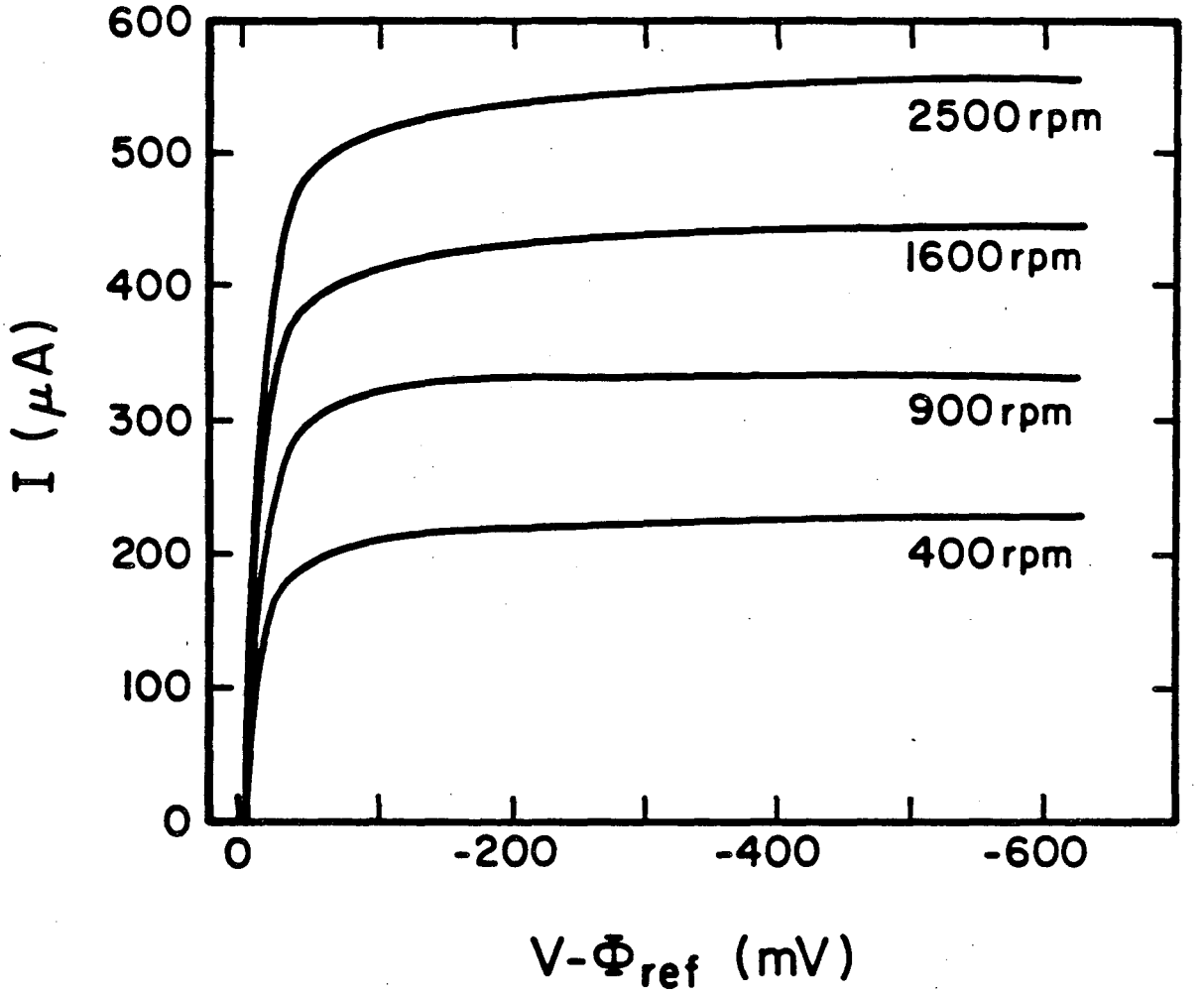


Figure 1-11. Sketch of rotating disk electrode apparatus.



XBL 848-3627

Figure 1-12. Polarization curves as a function of rotation speed. $c_b = 5\times10^{-7}$ mol /cm^3, $c_{Cl^-} = 3.8\times10^{-3}$ mol /cm^3.

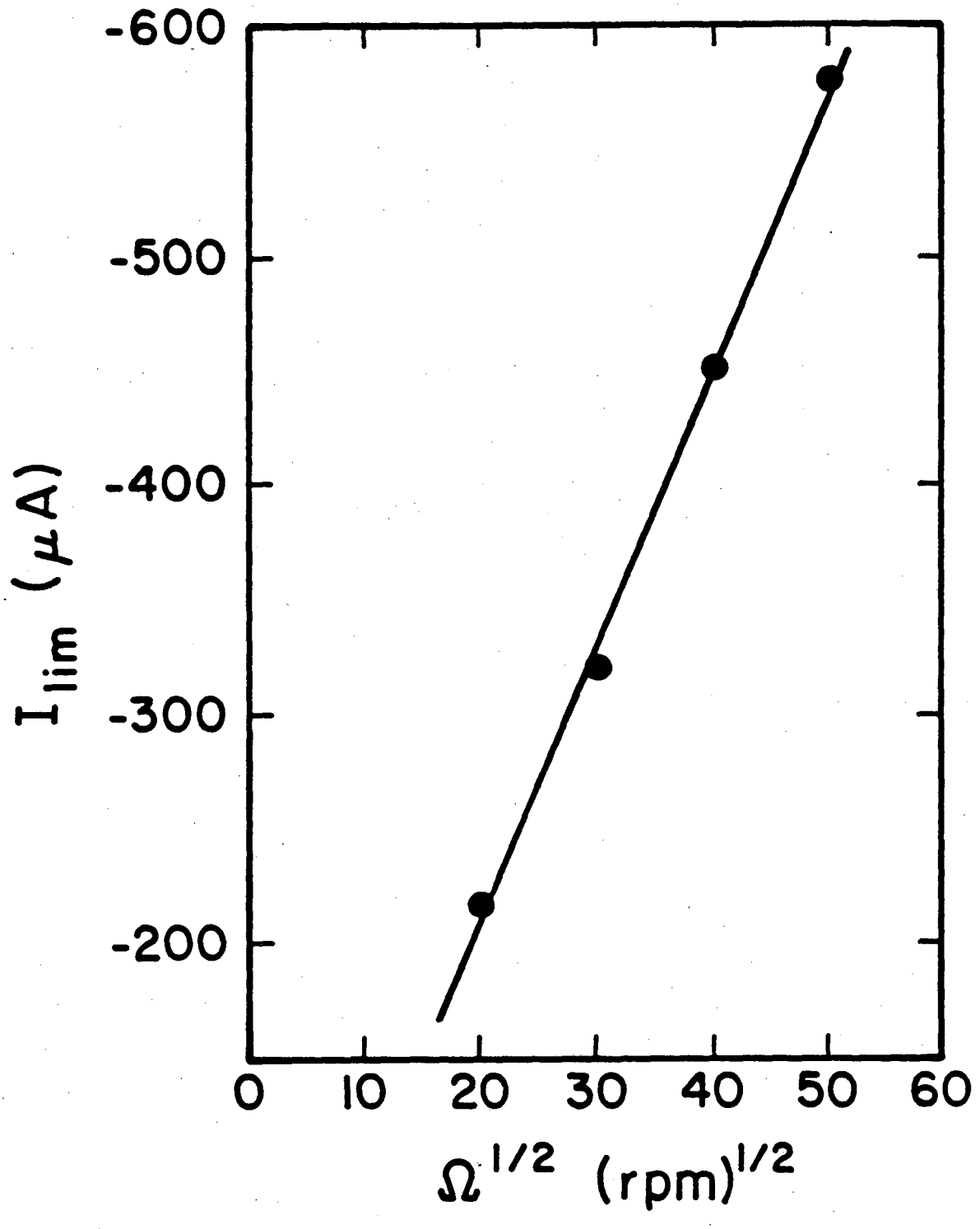


Figure 1-13. Limiting current versus square root of rotation speed.

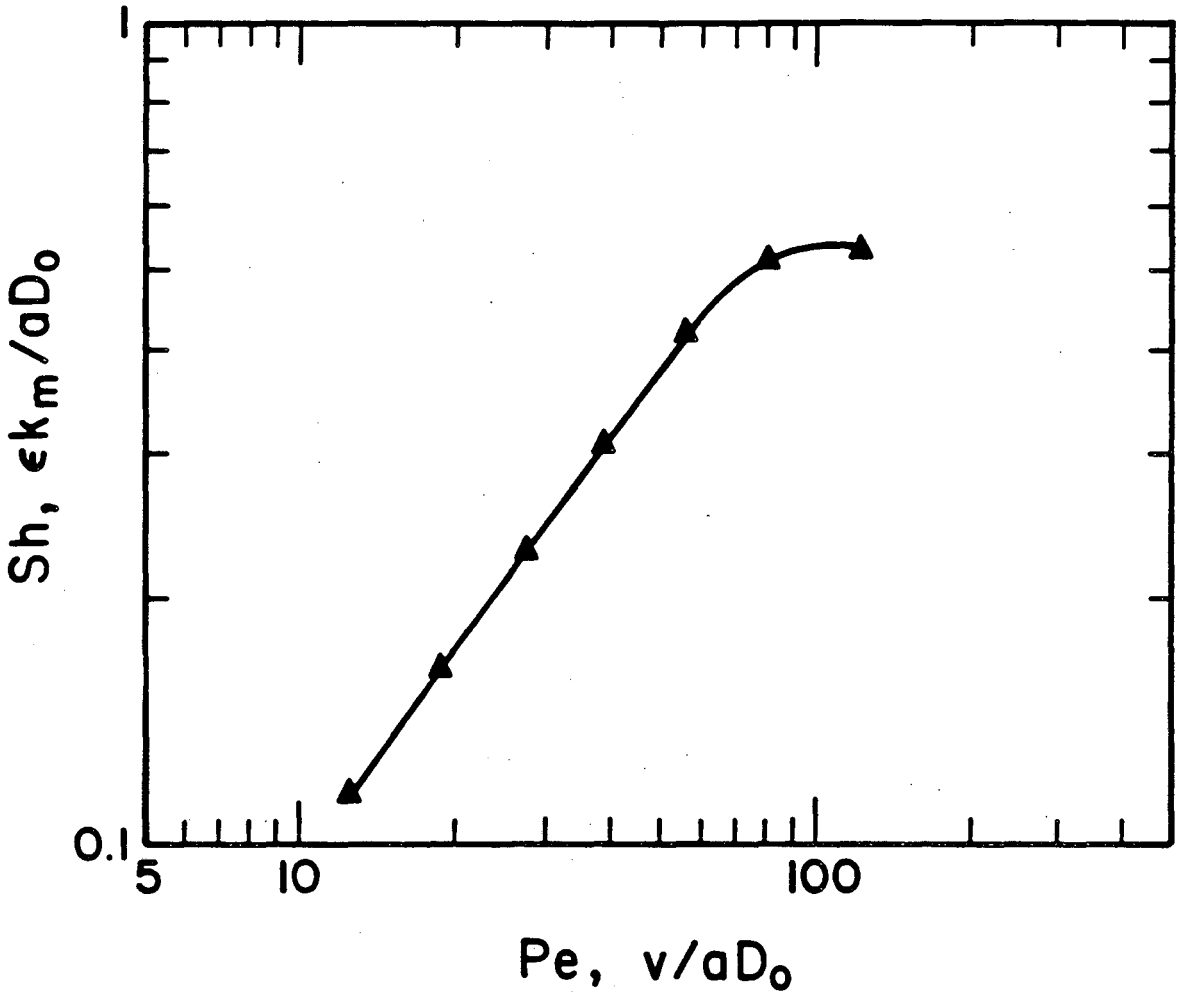


Figure 1-14. Effect of flowrate on the mass-transfer coefficient. Dimensionless Sherwood-number/Péclet-number plot. $L = 12.7$ cm, $a = 66$ cm^{-1}, $\epsilon = 0.97$, $D_o = 1.0\times10^{-5}$ cm^2/s.

$$Sh = \frac{\epsilon \bar{k}_m}{aD_o}, \tag{1-4}$$

is shown on the ordinate, and the dimensionless velocity, or Péclet number (Pe), defined as

$$Pe = Re \cdot Sc = \frac{v}{aD_o}, \tag{1-5}$$

is shown on the abscissa.

At low values of the Péclet number, the logarithm of the Sherwood number varies linearly with the logarithm of the Péclet number (with a slope of approximately one), and, as the Péclet number is increased, the Sherwood number gradually becomes independent of Péclet number. This behavior is characteristic of mass-transfer coefficients in packed beds, and it indicates a transition from a region where axial dispersion effects are important (low Péclet number) to a region where such effects can be neglected [21].

Since this correlation is general and depends only on the geometry of the RVC, it may be used in the scale-up and design of other reactors fabricated from RVC for metal-removal applications.

1.10. Comparison of Experiments to Model Predictions

One of the primary goals of this study is to check the applicability of the model of Trainham and Newman [14] to the study of flow-through porous electrodes made of RVC. To this end, figures 1-5, 1-6, 1-7, and 1-9 contain direct comparisons of experimental data to predictions from the model. In this section, an explanation is presented of the manner in which the fitting parameters for the model were chosen, and the quality of the fit is discussed. The comparison confirms the validity of the model and indicates that the exchange-current density for mercury deposition is much higher on solid glassy carbon than on RVC.

The presentation begins with an abbreviated description of the mathematical model, including the governing differential equations and definitions of the major parameters. The two major

fitting parameters are the exchange-current densities for the main reaction ($i_{oR,ref}$) and for the side reaction ($i_{oS,ref}$), which are chosen by fitting the porous-electrode data in figure 1-5. Curves on the other figures are calculated from the model with these parameters. Ohmic-drop data (figure 1-7) provide an independent check on the validity of the model, and the close agreement between the calculations and experiment there support the applicability of the model.

1.10.1. Trainham and Newman's Model

The model employed for this study is very similar to the one-dimensional, macrohomogeneous model of a flow-through porous electrode for metal-ion removal developed by Trainham and Newman. Additional details and a complete derivation of the governing equations may be found in [14]. A source listing of the computer program used for the calculations is contained in the second part of this dissertation.

The model, based on the earlier work by Newman and Tobias [22], considers the solution and electrode matrix as two superimposed continua, where the details of the internal pore structure can be effectively averaged. The electrolyte is assumed to be well supported and the side reaction to be concentration-independent. As a result, the problem, in dimensionless form, can be stated as a set of two coupled, nonlinear ordinary differential equations: a material balance on the local concentration of metal-ion reactant,

$$\frac{d\theta}{dy} = -J_R \ , \tag{1-6}$$

and a charge balance determining the local overpotential,

$$\frac{d}{dy}\left(\frac{1}{P_2}\frac{d\eta'}{dy} + \theta\right) = J_S \ . \tag{1-7}$$

θ represents metal-ion concentration, η' potential driving force, and y distance through the packed bed. J_R, a reaction-rate term for the main reaction (metal deposition), is defined as

$$J_R = \frac{\theta - P_1 \exp\left(\left(\frac{\alpha_{aR}}{\alpha_{cR}} + 1\right)\eta' \right)}{1 + \exp(\eta')}, \tag{1-8}$$

and J_S, a reaction-rate term for the side reaction (hydrogen evolution), is defined as

$$J_S = P_3 \exp\left(-\frac{\alpha_{cS}}{\alpha_{cR}}\eta' \right) \left[1 - P_4 \exp\left(\frac{\alpha_{aS} + \alpha_{cS}}{\alpha_{cR}}\eta' \right) \right]. \tag{1-9}$$

The concentration at the inlet to the reactor is fixed, as expressed by the boundary condition:

$$\text{at} \quad y = 0, \qquad \theta = 1. \tag{1-10}$$

Boundary conditions on the potential driving force depend on the placement of the counterelectrode and of the current collector. Two cases are considered here:

UD (upstream counterelectrode, downstream current collector), where,

at $y = 0$,

$$\frac{d\eta'}{dy} = P_5 I^*, \tag{1-11}$$

and, at $y = \alpha L$,

$$\frac{d\eta'}{dy} = -P_6 I^*, \tag{1-12}$$

and DU (downstream counterelectrode, upstream current collector), where,

at $y = 0$,

$$\frac{d\eta'}{dy} = P_6 I^*, \tag{1-13}$$

and, at $y = \alpha L$,

$$\frac{d\eta'}{dy} = -P_5 I^*. \tag{1-14}$$

P_1 characterizes the backward term of the main-reaction rate, P_2 the relative importance of ohmic resistance to mass-transfer resistance, P_3 the rate of the side reaction, and P_4 the backward term in the side-reaction rate. P_5 and P_6 represent the relative importance of the ohmic potential drop in the pore solution phase and the electrode matrix phase. The parameter α is the reciprocal of the penetration depth into the reactor, and I^* is the ratio of the actual current density to the current density that would exist if all of the metal-ion feed were completely reacted in the absence of a side reaction.

The model parameters and their definitions are summarized in tables 1-3 and 1-4. Although virtually identical to the definitions in [14], there is one important difference between the model equations and parameters shown here and those of Trainham and Newman. In this work, all of the effects of axial dispersion are included in the mass-transfer coefficient $\bar{k}_m$, whereas the original model employed local film coefficients k_m and considered the effect of dispersion separately by introducing an additional parameter, D'. By incorporating the dispersion effects directly into the mass-transfer coefficient, the present model allows the values of $\bar{k}_m$ from table 1-2 to be used without modification in the theoretical calculations. If, in [14], D' is taken to be zero and all instances of k_m are replaced by $\bar{k}_m$, the statement of the problem is identical to that shown here.

1.10.2. Model Parameters

Table 1-5 shows the parameter values chosen for the model presented in figures 1-5, 1-6, and 1-7. For the model results shown in all of the figures, physical property data and dimensional fitting parameters are the same. However, since operating conditions are not identical in all of the runs, values of some of the parameters for the model results shown in figure 1-9 differ from those shown in table 1-5. The changes are recorded in table 1-6.

The mass-transfer coefficients are taken from the results of the concentration measurements in table 1-2 and, therefore, are consistent with the data shown in figures 1-10 and 1-14. The operating conditions and the physical property data are all known (see table 1-1), and, therefore,

Table 1-3. Definitions of Special Quantities Derived for Model Calculations

Reciprocal of the Penetration Depth:

$$\alpha = \frac{a\bar{k}_m}{v}$$

Conductivity:

$$\kappa = \kappa_o \epsilon^{1.5}$$

Open-Circuit Potentials:

$$U_R = U_R^{\theta} - U_{re} + \frac{RT}{nF}\ln\left(\frac{c_{Rf}}{\rho_o}\right) - 2\frac{RT}{F}\ln\left(\frac{c_{Cl^-,f}}{\rho_o}\right)$$

$$U_S = U_S^{\theta} - U_{re} + \frac{RT}{F}\ln\left(\frac{c_{H^+,f}}{\rho_o}\right) - \frac{RT}{2F}\ln\left(P_{H_2,f}\right)$$

$$\Delta U = U_S - U_R$$

Exchange-Current Densities:

$$i_{oR,f} = i_{oR,ref}\left(\frac{c_{Rf}}{c_{R,ref}}\right)^{\gamma_R}\left(\frac{c_{Cl^-,f}}{c_{Cl^-,ref}}\right)^{\gamma_{R,Cl^-}}$$

$$i_{oS,f} = i_{oS,ref}\left(\frac{c_{H^+,f}}{c_{H^+,ref}}\right)^{\gamma_{S,H^+}}\left(\frac{P_{H_2,f}}{P_{H_2,ref}}\right)^{\gamma_{S,H_2}}$$

where,

$$\gamma_R = -s_R + \frac{\alpha_{cR}\, s_R}{n} \qquad \gamma_{R,Cl^-} = 2\,\alpha_{cR}$$

$$\gamma_{S,H^+} = 1 - \alpha_{cS} \qquad \gamma_{S,H_2} = \frac{\alpha_{cS}}{2}$$

Table 1-4. Definitions of Dimensionless Variables and Parameters

$$\theta = \frac{c_R}{c_{Rf}} \qquad \eta' = \frac{\alpha_{cR} F \eta}{RT} + \ln\left(- \frac{nF\bar{k}_m c_{Rf}}{s_R i_{oR,f}} \right)$$

$$y = \frac{a\bar{k}_m x}{v} \qquad I^* = \frac{s_R i}{nFvc_{Rf}}$$

$$P_1 = \left(- \frac{s_R i_{oR,f}}{nF\bar{k}_m c_{Rf}} \right)^{1 + \alpha_{aR}/\alpha_{cR}} \qquad P_2 = \frac{\alpha_{cR} nF^2 v^2 c_{Rf}}{s_R a\bar{k}_m RT} \left(\frac{1}{\kappa} + \frac{1}{\sigma} \right)$$

$$P_3 = - \frac{s_R i_{oS,f}}{nF\bar{k}_m c_{Rf}} \exp\left(\frac{\alpha_{cS} F \Delta U}{RT} \right) \left(- \frac{nF\bar{k}_m c_{Rf}}{s_R i_{oR,f}} \right)^{\alpha_{cS}/\alpha_{cR}}$$

$$P_4 = \left(- \frac{s_R i_{oR,f}}{nF\bar{k}_m c_{Rf}} \right)^{\frac{\alpha_{aS} + \alpha_{cS}}{\alpha_{cR}}} \exp\left(- \frac{F(\alpha_{aS} + \alpha_{cS})\Delta U}{RT} \right)$$

$$P_5 = - \frac{\sigma P_2}{\sigma + \kappa} \qquad P_6 = - \frac{\kappa P_2}{\sigma + \kappa}$$

the only fitting parameters for the model are the kinetic constants for the main reaction ($i_{oR,ref}$, α_{aR}, α_{cR}), the kinetic constants for the side reaction ($i_{oS,ref}$, α_{aS}, α_{cS}), and the partial pressure of hydrogen ($P_{H_2 f}$) in the feed to the reactor.

Table 1-5. Values of Parameters for Model Calculations

s_R =	-1	n =	2
α =	0.5440 cm^{-1}	$1/\alpha$ =	1.838 cm
$i_{oR,ref}$ =	3×10^{-7} A /cm^2	$c_{R,ref}$ =	5×10^{-7} mol /cm^3
		$c_{Cl^-,ref}$ =	3.8×10^{-3} mol /cm^3
α_{aR} =	1.4	α_{cR} =	0.6
γ_R =	0.7	γ_{R,Cl^-} =	1.2
$i_{oR,f}$ =	2.00×10^{-7} A /cm^2	c_{Rf} =	2.273×10^{-7} mol /cm^3
		$c_{Cl^-,f}$ =	4.3×10^{-3} mol /cm^3
$i_{oS,ref}$ =	2×10^{-6} A /cm^2	$c_{H^+,ref}$ =	1×10^{-3} mol /cm^3 (pH = 0)
		$P_{H_2,ref}$ =	1 atm
α_{aS} =	0.5	α_{cS} =	0.5
γ_{S,H^+} =	0.5	γ_{S,H_2} =	0.25
$i_{oS,f}$ =	5.32×10^{-10} A /cm^2	$c_{H^+,f}$ =	1×10^{-7} mol /cm^3 (pH = 4)
		$P_{H_2,f}$ =	5×10^{-7} atm
U_R =	-0.00536 V	U_S =	-0.295 V
ΔU =	-0.290 V		
P_1 =	2.865×10^{-6}	P_2 =	-0.2803
P_3 =	4.984×10^{-6}	P_4 =	133.93
P_5 =	0.2526	P_6 =	0.02775
αL =	6.909		

1.10.3. Effect of Side Reaction

The transfer coefficients for the side reaction (α_{aS}, α_{cS}) are chosen to be 0.5, in keeping with generally accepted mechanisms for hydrogen reduction. The parameters $P_{H_2,f}$ and $i_{oS,ref}$ determine the length of the limiting-current plateau as described by White and Newman [23]. Since there is virtually no H_2 gas in the feed, the value of $P_{H_2,f}$ is very uncertain. It is

Table 1-6. Additional Parameter Values for Figure 1-9

c_{Rf} (mol /cm^3)	3.125×10^{-7}	3.125×10^{-7}	3.125×10^{-7}
$i_{oR,f}$ (A /cm^2)	2.50×10^{-7}	2.50×10^{-7}	2.50×10^{-7}
U_R (V)	-0.00127	-0.00127	-0.00127
ΔU (V)	-0.294	-0.294	-0.294
v (cm/s)	0.0214	0.0128	0.00576
$\bar{k}_m$ (cm/s)	1.805×10^{-4}	1.157×10^{-4}	5.747×10^{-5}
α (cm^{-1})	0.5567	0.5989	0.6587
$1/\alpha$ (cm)	1.796	1.670	1.518
P_1	3.463×10^{-6}	1.525×10^{-6}	1.571×10^{-6}
P_2	-0.3161	-0.1750	-0.0719
P_3	3.719×10^{-6}	4.005×10^{-6}	4.500×10^{-6}
P_4	1.726×10^{2}	3.623×10^{2}	1.163×10^{3}
P_5	0.2848	0.1577	0.0647
P_6	0.03130	0.01733	0.007116
αL	7.070	7.606	8.366

important to note, however, that the side reaction occurs primarily in a Tafel range (backward term in the rate expression is small). Under these conditions, an increase in $P_{H_2,f}$ should have very little effect on the location of the side reaction. In fact, changing $P_{H_2,f}$ from 5×10^{-7} atm to 1×10^{-12} atm results in no shift in the location of the side reaction and a shift of only 4 mV in the open-circuit potential calculated from the model. As a result, the fitting parameter for the side reaction is primarily the exchange-current density $i_{oS,ref}$. It is chosen from a fit to figure 1-5, since no other experiments indicate a side reaction.

Little quantitative information about the side reaction can be determined from the small amount of data obtained in this study, but the experiments do show qualitatively that the side reaction is not a major consideration in the evaluation of the system for mercury removal. In this regard, the system studied here is a good candidate for checking the applicability of the model, since in the model the side reaction is considered only as a second-order effect. The relative unimportance of side reaction means that the behavior of the porous electrode depends primarily

on the kinetic parameters for the main reaction, and the choices of values for these parameters are discussed below.

1.10.4. Kinetic Constants for the Main Reaction

Transfer coefficients for the main reaction (α_{aR} , α_{cR}) are obtained directly from a fit of the linear sweep voltammograms on the rotating disk.† The exchange-current density ($i_{oR,ref}$), however, is chosen from a fit of the left side of the polarization curve in figure 1-5. The model fit to one of the voltammograms (from figure 1-12) on the disk electrode is shown in figure 1-15, both for the exchange-current density that best fits the rotating-disk data and for the value chosen for the model of the porous electrode. The exchange-current density obtained from the rotating disk experiments is 5000 times higher than that required to fit figure 1-5. This difference indicates that, although both the porous bed and the disk electrode are made of a glassy carbon, the activity of the internal surface of the RVC foam and the activity of the polished disk surface are considerably different. Although this is not surprising, it does point out the importance of experimenting directly with the porous-bed material prior to scale-up.

1.10.5. Consistency Check: Ohmic Potential Drop

If the conductivity of the electrode matrix is high relative to that of the solution (as in this study), resistance measurements can be used to determine kinetic parameters. In addition, the ohmic-drop measurements are independent of the polarization measurements, and, as a result, they provide a consistency check on the data obtained in this investigation. Thus, the close agreement between the model predictions and the experimental measurements of ohmic drop, as illustrated in figure 1-7, confirms the validity of the $i_{oR,ref}$ value chosen to fit figure 1-5.

† Methods for the numerical simulation of linear sweep voltammograms are described in more detail in the third part of this dissertation.

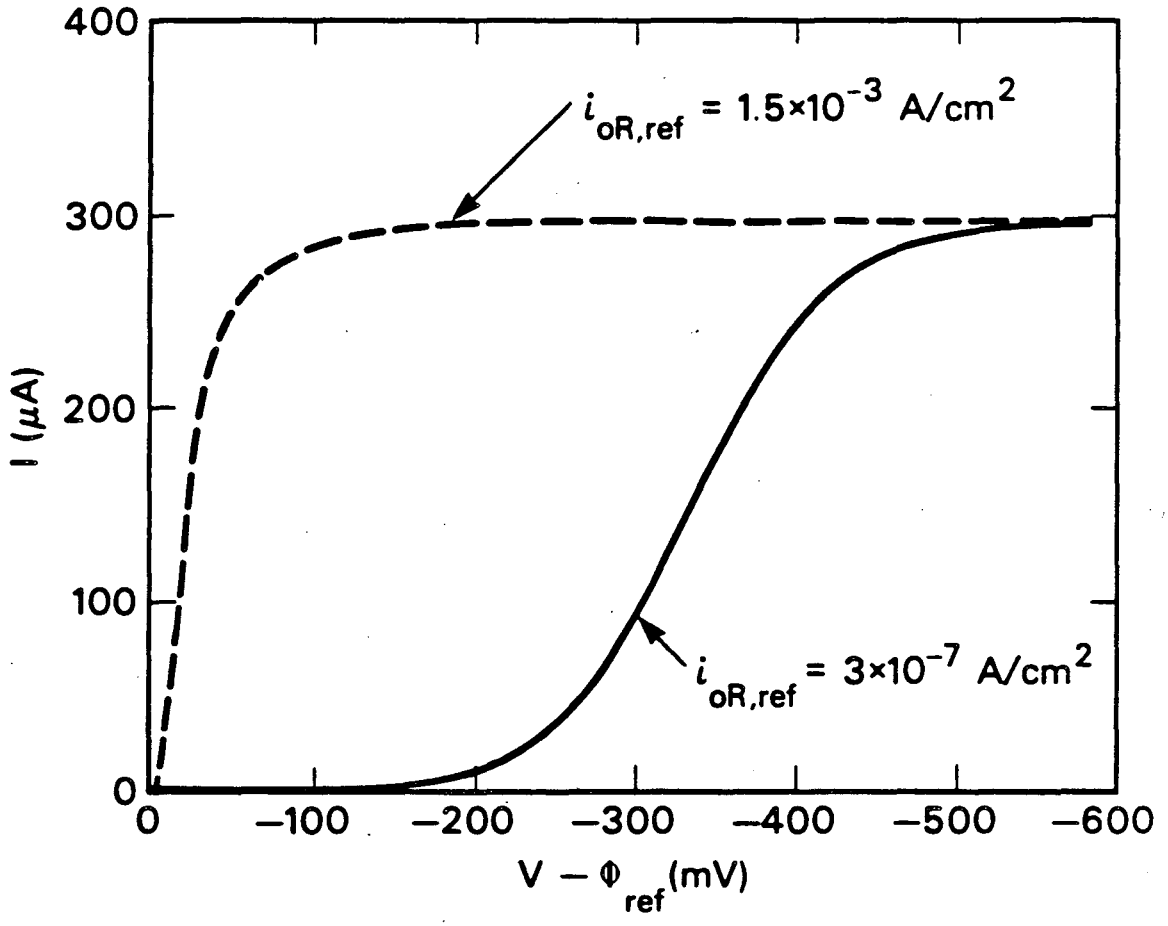


Figure 1-15. Effect of exchange-current density on model voltammograms. Dashed curve (best fit): $i_{oR,ref} = 1.5\times10^{-3}$ A /cm^2. Solid curve: $i_{oR,ref} = 3.0\times10^{-7}$ A /cm^2.

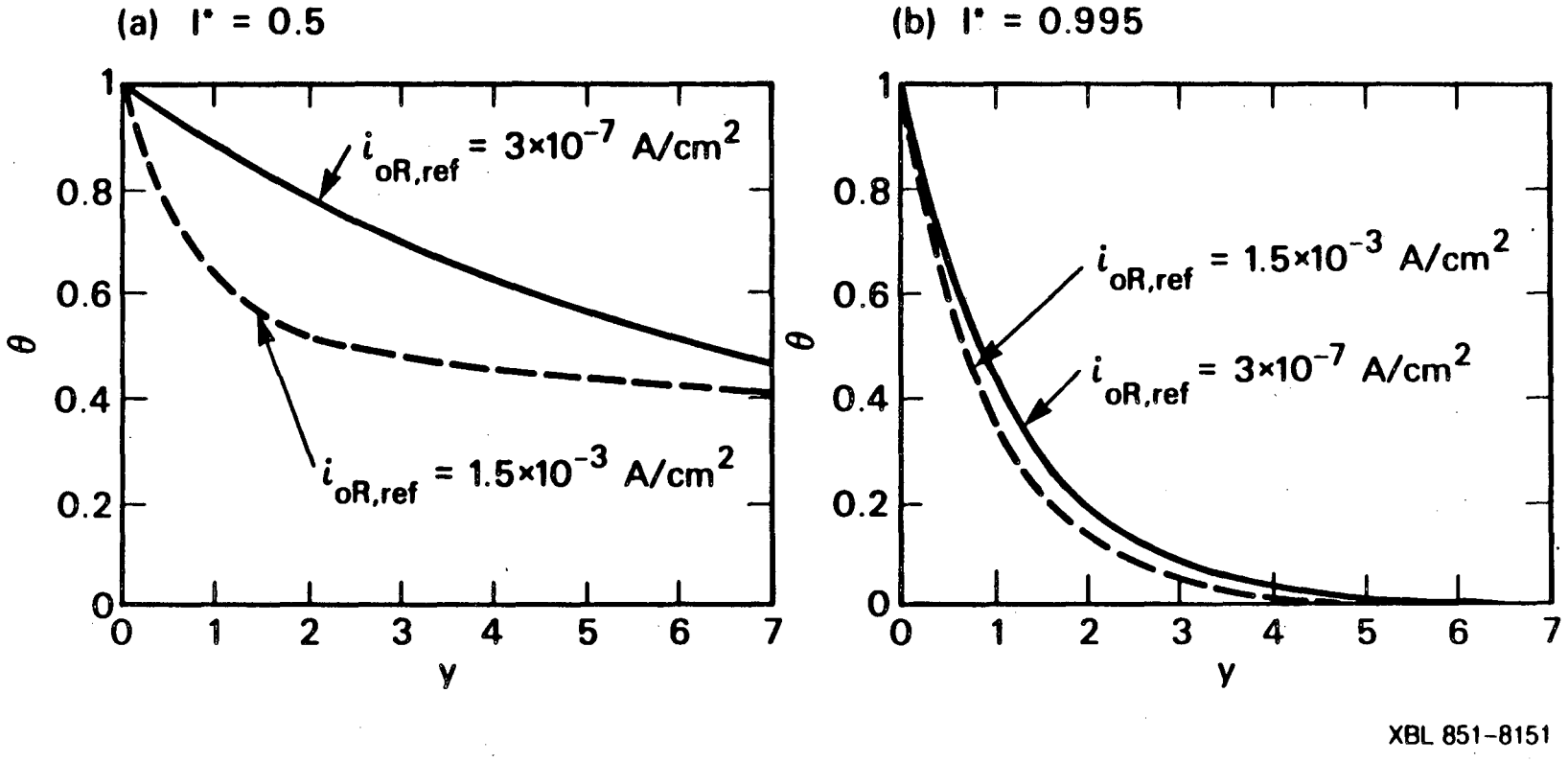
The reason for the sensitivity of the ohmic potential drop to kinetic parameters is illustrated in figure 1-16a for the case of upstream counterelectrode placement. At low fractions of limiting current, the ohmic drop measurements are strongly influenced by kinetics, since the value of the resistance is very sensitive to the shape of the upstream concentration profile. This sensitivity arises from the influence of the kinetic rate on the effective penetration depth. Since, in the absence of side reaction, the resistance is proportional to penetration depth, changes in exchange-current density can give rise to substantial differences in ohmic drop.

The ohmic drop measurements are particularly useful if the electrode is very long. If the electrode is very long, the catholyte exit can attain equilibrium for any moderate value of $i_{oR,ref}$, and, therefore, the value of $(V - \Phi_{ref})$ at the catholyte exit is fixed (by thermodynamics), independent of $i_{oR,ref}$. In that case, although polarization curves such as figure 1-5 provide no information at all regarding $i_{oR,ref}$, the effect of $i_{oR,ref}$ on ohmic drop (figure 1-7) remains unchanged. Thus, it is possible that large differences in ohmic drop can be seen even when only slight differences in polarization behavior are observable.

At limiting current, the polarization (driving force) is sufficiently large that the concentration profile is completely determined by mass transfer, regardless of the kinetic parameters. Under these conditions, illustrated in figure 1-16b, ohmic drop measurements provide no information about kinetics, but they do provide a check on the values of the mass-transfer coefficients. By comparing the results for upstream and downstream counterelectrode placement, the ratio of the penetration depth to the bed length can be determined (see figure 1-8). Hence, the agreement between model and experiment in figure 1-7 at high overpotential (limiting-current conditions) supports the mass-transfer coefficient correlation in figure 1-14.

1.11. Conclusions

This study has demonstrated that a flow-through porous electrode made of reticulated vitreous carbon can be a very effective device for the removal of mercury contamination in brine



XBL 851-8151

Figure 1-16. Effect of kinetics on ohmic drop measurements.

solutions. The experiments indicate that the mercury system is also an excellent candidate for the general study of porous electrodes, since the chemistry is simple, side reactions are relatively unimportant, and a reliable method for concentration measurement is available. A bench-scale experimental electrode has been used to demonstrate the effectiveness of the device, to illustrate the effect of counterelectrode placement on the ohmic potential drop, to determine an empirical correlation (in graphical form) of the effect of electrolyte velocity on the mass-transfer coefficient in the electrode, and to verify the applicability of a one-dimensional mathematical model of the porous electrode. A comparison of the exchange-current density for mercury deposition on RVC to that on a solid glassy carbon indicates that the local rate of mercury deposition is slower on RVC.

Acknowledgements

The author would like to thank Mr. Edward Brandenberger for suggesting the use of the gold-film analyzer for the mercury measurements. Special thanks are also due to Mr. Christopher Hofseth and to Mr. John Kelsey for their help with the rotating-disk experiments.

List of Symbols

a	Specific surface or interfacial area (per unit volume of the porous electrode), cm^2/cm^3
A	Area of rotating disk electrode, cm^2
$c_{Cl^-,f}$	Concentration of Cl^- ion in the catholyte feed to the porous electrode, mol $/cm^3$
$c_{Cl^-,ref}$	Reference concentration of Cl^-, mol $/cm^3$
c_b	Concentration of mercury in the bulk, far away from the rotating disk, mol/cm^3
$c_{H^+,f}$	Concentration of H^+ ion in the catholyte feed, mol $/cm^3$
$c_{H^+,ref}$	Reference concentration of H^+ ion, mol $/cm^3$
c_R	Concentration of main reactant, $HgCl_4^{-2}$, in the catholyte within the flow-through porous electrode, mol $/cm^3$
c_{Rf}	Concentration of main reactant, $HgCl_4^{-2}$, in catholyte entering the flow-through porous electrode, mol/cm^3
$c_{R,ref}$	Reference concentration of main reactant species, mol $/cm^3$
D_e	Diffusion coefficient of $HgCl_4^{-2}$ in the feed solution, cm^2/s
F	Faraday's constant, 96487 C/equiv
i	Superficial current density to the porous electrode, A $/cm^2$
$i_{eR,f}$	Exchange-current density for the main reaction at the feed concentrations, A $/cm^2$
$i_{eR,ref}$	Exchange-current density for the main reaction at the reference concentrations, A $/cm^2$
$i_{eS,f}$	Exchange-current density for the side reaction at the feed concentrations, A $/cm^2$
$i_{eS,ref}$	Exchange-current density for the side reaction at the reference concentrations, A $/cm^2$
I	Current, A

I_{lim}	Mass-transfer-limited current, A
I^*	Dimensionless applied current density (defined in table 1-4)
J_R	Dimensionless reaction-rate term for the main reaction
J_S	Dimensionless reaction-rate term for the side reaction
k_m	Local mass-transfer coefficient, cm/s
$\bar{k}_m$	Average mass-transfer coefficient, cm/s
L	Length of flow-through porous electrode, cm
n	Number of electrons transferred in the main electrode reaction
P_1	Parameter characterizing the backward term of the main reaction (defined in table 1-4)
P_2	Parameter characterizing the relative importance of ohmic resistance to mass-transfer resistance (defined in table 1-4)
P_3	Parameter characterizing the rate of side reaction (defined in table 1-4)
P_4	Parameter characterizing the backward term of the side reaction (defined in table 1-4)
P_5	Parameter representing the relative importance of ohmic drop in the solution phase (defined in table 1-4)
P_6	Parameter representing the relative importance of ohmic drop in the electrode matrix phase (defined in table 1-4)
Pe	Péclet Number, $Re \cdot Sc = v/aD_e$
$P_{H_2 f}$	Partial pressure of hydrogen gas in the feed to the porous electrode, atm
$P_{H_2 ref}$	Reference partial pressure of hydrogen gas, atm
R	Universal gas constant, 8.314 J/mol-K
Re	Reynolds Number, $v/a\nu$
Q	Flowrate, cm^3/min

S	Cross-sectional area of porous electrode, cm^2
Sc	Schmidt Number, ν/D_o
Sh	Sherwood Number, $\epsilon \bar{k}_m / aD_o$
s_R	Stoichiometric coefficient of the main reactant species
T	Temperature, K
v	Superficial fluid velocity, cm/s
U_{re}	Potential of a calomel reference electrode (containing saturated KCl) (with respect to a standard hydrogen electrode), V
U_R	Equilibrium potential of the main reaction at the feed concentrations (relative to U_{re}), V
U_R^{ℓ}	Standard electrode potential for the main reaction, $HgCl_4^{-2}/Cl^-/Hg$, (with respect to a standard hydrogen electrode), V
U_S	Equilibrium potential of the side reaction at the feed concentrations (relative to U_{re}), V
U_S^{ℓ}	Standard electrode potential for the side reaction, H^+/H_2, 0.0 V
ΔU	Difference in equilibrium potential between the side reaction and the main reaction at the feed concentrations, V
V	Potential of working electrode, V
$V - \Phi_{ref}$	Potential between the working electrode and a saturated calomel reference electrode in the catholyte exit stream, V
x	Distance through electrode, cm
y	Dimensionless distance through porous electrode (defined in table 1-4)
α	Reciprocal of the penetration depth into the electrode (defined in table 1-3)
αL	Dimensionless electrode length

α_{aR}	Anodic transfer coefficient for the main reaction
α_{aS}	Anodic transfer coefficient for the side reaction
α_{cR}	Cathodic transfer coefficient for the main reaction
α_{cS}	Cathodic transfer coefficient for the side reaction
γ_R	Exponent in the composition dependence of species R ($HgCl_4^{-2}$) in the expression for $i_{oR,f}$ (defined in table 1-3)
γ_{R,Cl^-}	Exponent in the composition dependence of Cl^- in the expression for $i_{oR,f}$ (defined in table 1-3)
γ_{S,H^+}	Exponent in the composition dependence of H^+ in the expression for $i_{oS,f}$ (defined in table 1-3)
γ_{S,H_2}	Exponent in the composition dependence of H_2 in the expression for $i_{oS,f}$ (defined in table 1-3)
ϵ	Porosity or void-volume fraction
η'	Dimensionless potential driving force, (defined in table 1-4)
θ	Dimensionless concentration (defined in table 1-4)
κ	Effective conductivity of electrolyte within the porous electrode, mho/cm
κ_o	Conductivity of feed solution outside of the electrode, mho/cm
μ	Viscosity, g/cm-s
ν	Kinematic viscosity, μ/ρ_o, cm^2/s
ρ_o	Solvent density, kg/cm^3
σ	Effective conductivity of the solid matrix, mho/cm
Φ_{ref}	Reference potential, V
$\Phi_{downstream}$	Potential in solution exiting electrode, V
$\Phi_{upstream}$	Potential in solution entering electrode, V
Ω	Angular velocity, rad/s

References

1. Douglas N. Bennion and John Newman, "Electrochemical removal of copper ions from very dilute solutions," *Journal of Applied Electrochemistry*, **2** (1972), 113-122.

2. Richard Alkire and Brian Gracon, "Flow-Through Porous Electrodes," *Journal of the Electrochemical Society*, **122** (1975), 1594-1601.

3. John Van Zee and John Newman, "Electrochemical Removal of Silver Ions from Photographic Fixing Solutions Using a Porous Flow-Through Electrode," *Journal of the Electrochemical Society*, **124** (1977), 706-708.

4. M. Enriquez-Granados, G. Valentin, and A. Storck, "Electrochemical Removal of Silver Using a Three-Dimensional Electrode," *Electrochimica Acta*, **28** (1983), 1407-1414.

5. James Trainham and John Newman, "The Removal of Lead Ions from Very Dilute Solutions Using a Porous Flow-Through Electrode," *Inorganic Materials Research Division Annual Report 1973*, 51-53. Lawrence Berkeley Laboratory, University of California, April, 1974, LBL-2299.

6. Gary George Trost, *Applications of Porous Electrodes to Metal-Ion Removal and the Design of Battery Systems*, dissertation, University of California, Berkeley (1983), LBL-16852.

7. J. Wang and H. D. Dewald, "Deposition of Metals at a Flow-Through Reticulated Vitreous Carbon Electrode Coupled with On-Line Monitoring of the Effluent," *Journal of the Electrochemical Society*, **130** (1983), 1814-1818.

8. A. T. Kuhn, "Antimony removal from dilute solutions using a restrained bed electrochemical reactor," *Journal of Applied Electrochemistry*, **4** (1974), 69-73.

9. James Arthur Trainham, *Flow-Through Porous Electrodes*, dissertation, University of California, Berkeley (1979). LBL-9565.

10. Michael Matlosz and John Newman, "Use of a Flow-Through Porous Electrode for Removal of Mercury from Contaminated Brine Solutions," *Proceedings of the Symposium on Transport Processes in Electrochemical Systems* (The Electrochemical Society, volume 82-10, 1982). 53-63.

11. John Newman and William Tiedemann, "Flow-through Porous Electrodes," *Advances in Electrochemistry and Electrochemical Engineering*, **11**, Heinz Gerischer and Charles W. Tobias, editors (New York: John Wiley and Sons, Inc., 1978), 353-438.

12. Bertrand P. Scholder, "Etude d'une électrode à empilement de treillis pour la récupération des métaux en solutions diluées,", dissertation, Swiss Federal Institute of Technology (ETH), Zurich (1982).

13. Roman E. Sioda and Kenneth B. Keating, "Flow Electrolysis with Extended-surface Electrodes,", Allen J. Bard, editor, *Electroanalytical Chemistry, A Series of Advances*, Allen J. Bard, editor (New York: Marcel Dekker, Inc., 1982), **12**, 1-51.

14. James A. Trainham and John Newman, "A Flow-Through Porous Electrode Model: Application to Metal-Ion Removal from Dilute Streams," *Journal of the Electrochemical Society*, **124** (1977), 1528-1540.

15. James A. Trainham and John Newman, "The Effect of Electrode Placement and Finite Matrix Conductivity on the Performance of Flow-Through Porous Electrodes," *Journal of the Electrochemical Society*, **125** (1978), 58-68.

16. P. J. Murphy, "Determination of Nanogram Quantities of Mercury in Liquid Matrices by a Gold Film Mercury Detector," *Analytical Chemistry*, **51** (1979), 1599-1600.

17. *Handbook of Chemistry and Physics*, 57 th edition, Robert C. Weast, editor (Cleveland, Ohio: CRC Press, Inc., 1976).

18. Joseph Wang, "Reticulated Vitreous Carbon -- A New Versatile Electrode Material," *Electrochimica Acta*, **26** (1981), 1721-1726.

19. Peter S. Fedkiw, *Mass Transfer Controlled Reactions in Packed Beds at Low Reynolds Numbers*, dissertation, University of California, Berkeley (1978), LBL-8509.

20. B. Levich, "The Theory of Concentration Polarization," *Acta Physicochimica U.R.S.S.*, **17** (1942), 257-307.

21. J. P. Sorensen and W. E. Stewart, "Computation of Forced Convection in Slow Flow Through Ducts and Packed Beds -- I Extensions of the Graetz Problem," *Chemical Engineering Science*, **29** (1974), 811-817.

22. John S. Newman and Charles W. Tobias, "Theoretical Analysis of Current Distribution in Porous Electrodes," *Journal of the Electrochemical Society*, **109** (1962), 1183-1191.

23. Ralph White and John Newman, "Simultaneous Reactions on a Rotating-Disk Electrode," *Journal of Electroanalytical Chemistry*, **82** (1977), 173-186.

Part 2. Solving One-Dimensional Boundary-Value Problems with BandAid: A Functional Programming Style and a Complementary Software Tool

2.1. Summary

This manuscript describes a functional style for writing computer programs to solve one-dimensional boundary-value problems, and it presents a software tool, BandAid, that supports programming in this style. The programming methodology described here permits a static, self-documenting description of the differential equations to be placed directly in the program listing, thereby increasing the reliability of new programs and simplifying the maintenance and extension of existing ones. BandAid creates a finite-difference representation of the problem described in the program listing, and it uses the banded-matrix algorithm of Newman [1,2,3] to solve the resulting set of algebraic equations.

The writing style is introduced by discussing the use of BandAid to solve a set of problems representative of typical applications. Then, once the basic principles of the programming style have been described, the numerical method is explained in detail, and, finally, a more flexible (but more complicated) routine, BandShell, is presented. Appendices include annotated listings of the BandAid and BandShell procedures as well as information concerning their implementation with the VAX/VMS operating system.

2.2. Introduction and Overview

This project was begun, in 1981, in an attempt to automate the determination of concentration and potential profiles in one-dimensional models of electrochemical systems. In general, determining these profiles involves solving boundary-value problems composed of sets of coupled, nonlinear, ordinary differential equations. Although a powerful finite-difference technique, Newman's BAND algorithm [1,2,3], had been developed to solve these equations, little general-

purpose software was available to aid preparation of the computer programs necessary to use it. In addition, maintenance of existing programs was difficult, since the differential equations being solved were not easily deduced from the code produced to solve them.

The goals of the project were, first, to identify a systematic approach to solving ordinary differential equations by Newman's method, and, second, to create a software tool that would facilitate reliable implementation of that approach for any arbitrary equation specification.

The results of the project consist of a recommendation for a specific writing style and of a software tool that supports the style. The writing style is a functional, or applicative, style [4]; the major portion of the calling program is written using function subprograms and constants only, and variables and procedure subprograms are avoided. Although limited, this approach is advantageous, since it permits the mathematical nature of the problem to be reflected directly in the computer program, and, therefore, the code generated for a particular application can be determined directly from the differential equations to be solved.

The software tool, BandAid, accepts code written in the functional style, interprets it, and uses the BAND algorithm to calculate a numerical solution to the problem described. The procedure is written in ANSI/IEEE-standard Pascal, and it is small and portable. Furthermore, the class of ordinary differential equations supported is general enough that BandAid can be used for many one-dimensional modeling problems in electrochemistry as well as in other physical sciences.

The purpose of this document is to explain both the writing style and its use in preparing computer programs that employ the BandAid routine. The intent throughout is to be as complete as possible, so that this single document can provide all of the information needed by a BandAid user. Whenever possible, examples are used to illustrate important points, and possible pitfalls and limitations of the program are indicated where they can be anticipated. As with any software package, the ultimate authority on what the program does or does not do is the source code itself, and the reader is encouraged to examine the listing of the BandAid procedure (provided in the appendix) whenever questions or ambiguities arise. In order to guarantee the accuracy of the list-

ings provided in this document, the text of all of the programs has been typeset directly from the machine-readable code used to check the examples. Nevertheless, a few bugs may have slipped through, and the reader should beware.

The BandAid procedure and all programs presented in this document are written in the Pascal programming language. Pascal was chosen, since it encourages modular, top-down programming design, compilers supporting numerical computation with Pascal are widely available, and programs written carefully in Pascal are generally easy to read and maintain. Naturally, those readers interested in using BandAid for non-trivial extensions to the examples provided should be familiar with Pascal programming, but readers familiar with any high-level programming language should be able to follow the general discussion.

Cooper and Clancy's *Oh! Pascal* [5] contains an excellent introduction to the language, and Cooper's book *Standard Pascal* [6] provides an explanation of the International Organization for Standardization (ISO) version of Pascal approved by the American National Standards Institute (ANSI) and the Institute of Electrical and Electronics Engineers (IEEE). Readers interested in the development and philosophy of the language should consult the original work by Wirth [7,8,9] and by Jensen and Wirth [10]. Discussion of the advantages and disadvantages of Pascal may be found in references [11] and [12]. Formatting conventions and general rules for preparation of the source code found in this document follow the recommendations in [13] and [14].

2.3. Organization and Subprogram Hierarchy

Figure 2-1 illustrates the overall subprogram hierarchy, indicating the relationships among the major program segments. Although the user program communicates only with the BandAid package, as shown at the top of the diagram, BandAid is not a self-contained unit. Rather, it is an interface to the more general (but more complicated) BandShell code, which, for some applications, the user program may need to access directly. Nested within BandShell is BandCore, the

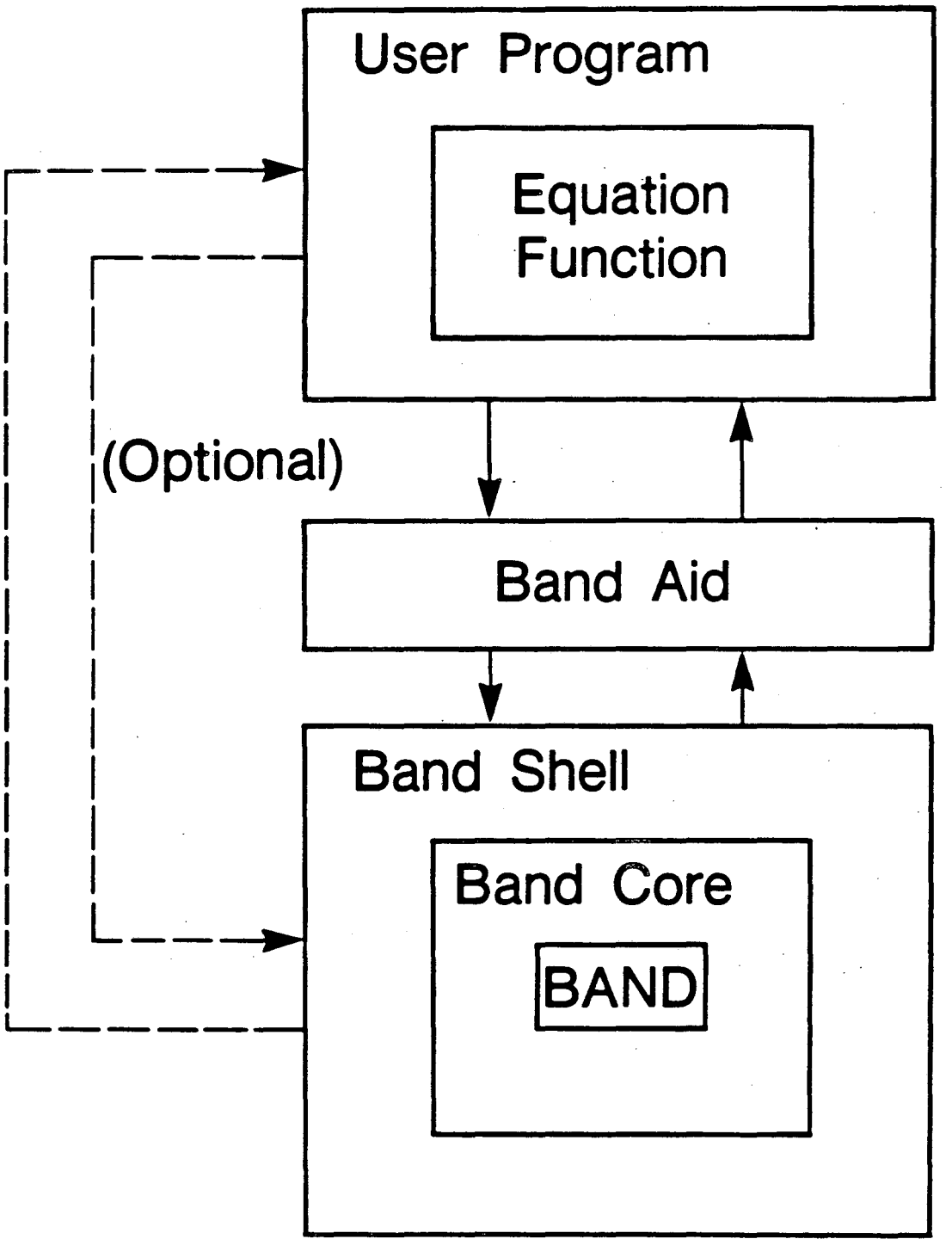


Figure 2-1. General Organization and Subprogram Hierarchy.

code for Newman's BAND algorithm, which performs most of the computational work.

The solution of differential equations with BandAid requires that the user write a main program containing a functional description of the equations to be solved. This description, written as a function subprogram, is then passed to BandAid, where it is used during the determination of a converged solution. Although other information must be sent to BandAid as well, the key to the method is the structure of this function subprogram. Consequently, the first part of this document is directed primarily toward construction of these functions. Since the basic principles involved may be obscured by presenting too much detail, the first illustrative examples employ only the simpler BandAid procedure, and a discussion of BandShell is postponed until later. Although BandAid is not as flexible as BandShell, it is still useful for many applications that do not require the complete BandShell procedure, and, in most cases, using BandAid is identical to using BandShell with default values for many of the parameters already selected.

2.4. A First Example: Concentration Distributions in a Catalytic Reactor

2.4.1. Problem Statement

Consider the simultaneous diffusion, convection, and reaction of two chemical species, A and B, at steady state in a packed-bed, catalytic reactor of length L. The catalyzed chemical reaction is reversible and can be represented by the following equation:

$$2A \rightleftharpoons B \ .$$

If the reaction is an elementary step, then the rate of disappearance of species A, R_A, can be determined directly from the stoichiometry as

$$R_A = k_f c_A^2 - k_b c_B \ , \tag{2-1}$$

where c_A and c_B are the concentrations of the species and k_f and k_b are rate constants.

A material balance on each species yields the following set of coupled, nonlinear, ordinary differential equations:

$$D_A \frac{d^2 c_A}{dx^2} - v\frac{dc_A}{dx} - R_A = 0 , \tag{2-2}$$

and

$$D_B \frac{d^2 c_B}{dx^2} - v\frac{dc_B}{dx} + \frac{1}{2}R_A = 0 , \tag{2-3}$$

where v is the average fluid velocity and D_A and D_B are diffusion coefficients. Furthermore, if the inlet to and the outlet from the reactor are packed with inert material, the boundary conditions may be represented by the classical Wehner-Wilhelm-Danckwerts conditions [15,16]:

at $x = 0$,

$$\begin{aligned} -D_A \frac{dc_A}{dx} + v(c_A - c_{A,o}) &= 0 , \\ -D_B \frac{dc_B}{dx} + v(c_B - c_{B,o}) &= 0 , \end{aligned} \tag{2-4}$$

and, at $x = L$,

$$\frac{dc_A}{dx} = \frac{dc_B}{dx} = 0 ,$$

where $c_{A,o}$ and $c_{B,o}$ are the concentrations of species A and B in the feed to the reactor.

2.4.2. Sample Program

Figure 2-2 shows program CatReac, the calling program for this first example. It contains a function subprogram representing the differential equations to be solved along with the corresponding boundary conditions. In addition to the function subprogram, it lists parameter declarations (in the **const** and **var** statements) and a program body (near the **end**) containing the BandAid call. The next few sections describe the program, beginning with the declarations at the

Figure 2-2. Text of Program CatReac

```
[INHERIT('AidMod.pen','CatIO.pen')]

program CatReac( input, output );

const   n = 2;
        Tolerance = 1e-6;

var     DA, DB, cAo, cBo, v, L, kf, kb : RealNumber;
        jMax, j, ItMax, It, CPUTime : integer;
        Guess, Result, Residual : ValueArray;


function Equation (
                    i, j : integer;

                    x, h : RealNumber;

                    var NewResult : ValueArray;

                    function cInterp ( k : integer;
                                       x : RealNumber;
                                       var Result : ValueArray ) : RealNumber;

                    function dnFdxn ( n : integer;
                                      function F ( x : RealNumber ) : RealNumber;
                                      x : RealNumber                ) : RealNumber

                                                                    ) : RealNumber;

        function cA( x : RealNumber ) : RealNumber;
        begin   cA := cInterp(1,x,NewResult)    end;

        function cB( x : RealNumber ) : RealNumber;
        begin   cB := cInterp(2,x,NewResult)    end;

        function ddx( function G( x : RealNumber ) : RealNumber;
                          x : RealNumber                    ) : RealNumber;
        begin
                ddx := dnFdxn(1,G,x)
        end;

        function d2dx2( function G(x : RealNumber ) : RealNumber;
                            x : RealNumber                  ) : RealNumber,
        begin
                d2dx2 := dnFdxn(2,G,x)
        end;

        function RA( x : RealNumber ) : RealNumber;
        begin   RA := kf*cA(x)*cA(x) - kb*cB(x) end;

        function DiffEQ( i : integer; x : RealNumber ) : RealNumber;
        begin
                case i of

                1:  DiffEQ := DA*d2dx2(cA,x) - v*ddx(cA,x) - RA(x);   { = 0 }
                2:  DiffEQ := DB*d2dx2(cB,x) - v*ddx(cB,x) + RA(x)/2  { = 0 }
```

```
                end   { i cases }

        end;

        function BC1( i : integer ) : RealNumber;
        begin
                case i of

                1:  BC1 := -DA*ddx(cA,0) + v*( cA(0) - cAo );   { = 0 }
                2:  BC1 := -DB*ddx(cB,0) + v*( cB(0) - cBo )    { = 0 }

                end   { i cases }
        end;

        function BC2( i : integer ) : RealNumber;
        begin
                case i of

                1:  BC2 := ddx(cA,L);    { = 0 }
                2:  BC2 := ddx(cB,L)     { = 0 }

                end   { i cases }
        end;


begin   { body of Equation }

        if ( j = 1 ) then
                Equation := BC1(i)

        else if ( j > 1 ) and ( j < jMax ) then
                Equation := DiffEQ(i,x)

        else if ( j = jMax ) then
                Equation := BC2(i)

end;   { Equation }


begin   { body of CatReac }

        ReadAndPrintParameters( DA, DB, cAo, cBo, v, L, kf, kb, jMax, ItMax );

        for j := 1 to jMax do
        begin
                Guess[1,j] := cAo;
                Guess[2,j] := cBo
        end;

        BandAid( n, jMax, ItMax, It, CPUTime,
                        L, Tolerance, Equation, Guess, Result, Residual );

        PrintOut( Result, jMax, It, CPUTime, L )

end.  { CatReac }
```

top[†] and proceeding down to the program body.

2.4.3. Special Data Types and Parameter Declarations

Two special data types, RealNumber and ValueArray, are employed throughout. A RealNumber is a floating-point number (usually double-precision), and a ValueArray is a matrix for storage of the values of the dependent variables. These types are defined externally, in a scope global to the program, and they are used in the same way as the standard data types integer, real, boolean, and char.

The **const** and **var** statements declare a number of important identifiers for the program, as follows: n is the number of equations and unknowns (two, in this case); Tolerance is a convergence tolerance; jMax is the number of mesh points; ItMax is a limit on the number of iterations; It is the actual number of iterations required; and L is the size of the domain. Guess stores the values of an initial guess of the solution, Result stores the answer returned by BandAid, and Residual stores the values of the equations at the final conditions. The meshpoint index j is used in the program body, and the remaining identifiers are parameters for the model.

Those identifiers declared as **const** are defined at compilation time, and those declared as **var** are read from the input or are assigned values during execution. With the exception of It, Result, and Residual, all remain unchanged during the execution of BandAid.

[†] The first line contains the non-standard INHERIT directive of VAX-11 Pascal. This directive alerts the compiler to the existence of other, separately-compiled modules (specifically AidMod and CatIO) that will be linked to the user program. Separate compilation of modules is permitted in VAX-11 Pascal, although its use is non-standard; other systems will be different. The directive is included in the listing here so that the program can be compiled exactly as written without modification. A more complete discussion of the use of modules in VAX-11 Pascal is found later in the appendix.

2.4.4. The Parameter List of the Equation Function

The name of the function subprogram describing the equations to be solved is Equation, and the structure of this subprogram illustrates the basic principles underlying the functional style: the description of the differential equations to be solved is made solely in terms of constants, functions, and composite functions (i. e., functions of functions). Variables are not employed as in conventional programming, but, rather, only as parameters to the functions. This is what is meant by a functional style.

Equation's parameter list contains two primitive functions (cInterp and dnFdxn) that form the basis of the problem description. All of the local functions declared in Equation will be defined directly in terms of these primitives (as, for example, cA or cB), or they will be defined in terms of composite functions of those functions (as, for example, RA or DiffEQ). These primitives, cInterp and dnFdxn, form the link between BandAid and the user's description of the problem to be solved. When the BandAid procedure requires information about the differential equations, it calls the Equation function, replaces the formal parameter list (i, j, x, h, NewResult, cInterp, dnFdxn) with its own actual parameters, and then interprets the function described therein. This process is performed repeatedly as the BandAid procedure searches for the solution to the equations.

The user program defines the Equation function over a continuous domain, x, and, for coupled problems, it distinguishes the individual equations by **case** statements in the function body. The first four arguments of Equation are simple value parameters: the particular equation under consideration (the i^{th} equation) is indicated by the parameter i (an integer), the mesh point (the j^{th} point) is indicated by the parameter j (an integer), a location in the domain by x (a floating-point number), and the mesh size by h (a floating-point number). The fifth parameter, NewResult (a ValueArray), contains the values of the dependent variables in BandAid.[†]

[†] Notice that the declaration is preceded by the keyword **var**. This indicates that the parameter is passed by name, rather than by value. Since the array is typically quite large, this provides a considerable improvement in efficiency.

The cInterp function, Equation's sixth parameter, takes three parameters of its own. The first, k (an integer), represents a particular dependent variable (the k^{th} dependent variable), the second, x (a floating-point number), represents a location in the domain, and the third, Result (a ValueArray), represents an array of values of the dependent variables at discrete mesh points along the domain.[††] The Result matrix contains values of the variables along the domain, and the cInterp function interpolates (linearly) between two of these to determine the value of a variable at a point x. Thus, the cInterp function provides a conversion from the discrete representation of the equations required by the numerical method to the continuous representation of the equations supplied by the user.

Equation's final parameter is the dnFdxn function. It takes three parameters, one of which is itself a function. The first parameter, n (an integer), represents the order of the differentiation (0, 1, or 2). The second parameter, F (a function subprogram), represents any function defined continuously over the x domain. Finally, the third parameter, x (a floating-point number), represents the point in the domain where the derivative is to be taken.

2.4.5. Local Declarations and the Body of the Equation Function

Development of a typical Equation function proceeds as follows:

First, all of the dependent variables are defined and given names by suitable redefinition of the cInterp functions. In the example shown, these functions are called cA(x) and cB(x). These new functions are now functions of a single variable x, and they may be used in the dnFdxn function, or they may be employed in the definition of composite functions. For example, the reaction rate at a point, RA(x), is a composite defined in terms of cA(x) and cB(x). Of course, functions may also be composed from the dnFdxn primitive. In the program fragment, for example, a first-derivative operator, ddx(G,x), and a second-derivative operator, d2dx2(G,x), are defined from dnFdxn, where G(x) is any function or composite defined continuously over the domain. Once all

[††] Result is also a var parameter. (See above.)

of these intermediate functions and composite functions are declared, Equation itself is defined in terms of them in the function body. As shown in figure 2-2, a function (like DiffEQ) normally provides a **case** statement to indicate exactly which differential equation represents a given number in the set. The equation should be written as a residual, which will be equal to zero for a converged solution, since this is the convention assumed by BandAid. It is recommended, therefore, to emphasize this fact by the comment

$$\{ = 0 \}$$

following the equation definition.

In general, each mesh point in the domain may be represented by a different set of equations, but, normally, only three are needed: differential equations (DiffEQ), boundary conditions at $x = 0$ (BC1), and boundary conditions at $x = L$ (BC2).

Although the content will vary from application to application, every Equation subprogram declared for use with BandAid has the same fundamental character. All of the intermediate functions declared simply define identifiers within the subprogram. There is no action; no assignments are made to storage locations within the computer. In fact, the Equation function does not *do* anything, nor does any function declared within it affect any aspect of the program outside of its scope. The only non-local identifiers used are those representing model constants† (like D_A or k_f) that remain unchanged throughout execution. In short, the function is purely descriptive, the description is local, and there are no side effects. This static, insulated property is key to the method.

Naturally, the representation of the equations shown here is not the only possibility. A rather large number of functions has been defined to enhance readability and to illustrate the programming style, and, ultimately, each programmer will develop his own personal taste. For suc-

† The term *constant* here does not necessarily mean that the identifiers are declared as **const**. They may, in fact, be declared as **var**, particularly if the BandAid call is only a part of a larger program. All that is required is that they remain unchanged during the execution of the BandAid procedure.

cessful use of BandAid, the user program need not look exactly like the example. All that is required is that the Equation subprogram conform to the basic structure of the functional style.

2.4.6. The Program Body and Sample Output

The program body prints the parameters, sets a guess of the problem solution, calls the BandAid routine, and prints the results. Just as the types RealNumber and ValueArray are defined outside of the scope of the user program, the procedures BandAid, ReadAndPrintParameters, and PrintOut are assumed to be defined externally.† (Their texts are shown in the Appendix.) Figure 2-3 shows sample output for a typical set of parameters.

2.4.7. Discussion

The program in figure 2-2 illustrates the basic principle of the programming style, namely, that the specification of the equations to be solved will be composed, as much as possible, of constants (such as DA or kf), functions (such as cA or d2dx2), and functions of functions (such as RA) (i. e., it contains no variables). This approach renders the description of the equations static, and, as a result, the statement of the problem is not affected by aspects of the program that are changing during execution. Furthermore, since (with the exception of constants) only local identifiers are employed, the description is insulated from side effects in other parts of the user program as well as from side effects in the BandAid package. By contrast, a traditional programming style does not distinguish between the description of the problem and the machinery involved in the problem solution: variables are employed directly in the user program. This has the unfortunate consequence that the description of the equations is dynamic (i. e., it depends upon the state of machine storage) and, therefore, it varies during program execution.

---

† As noted in section 2.2.2, these external definitions are permitted by the independent-compilation mechanism of VAX-11 Pascal. Specific features of that mechanism are described later in the appendix.

Figure 2-3. Sample Output from CatReac

CatReac Program

Parameters

DA =	1.0000E-06	cm2/sec	
DB =	1.0000E-06	cm2/sec	
cAo =	0.20000	mol/cm3	
cBo =	0.050000	mol/cm3	jMax = 26
v =	0.0050000	cm/sec	ltMax = 12
L =	10.000	cm	
kf =	0.100000	cm3/mol-sec	
kb =	1.0000E-03	1/sec	

Profile Listing

Number of iterations = 7
Execution time = 10410 milli-seconds

j	x	cA	cB
1	0.00000	0.1999315	0.05003427
2	0.40000	0.09745880	0.1012706
3	0.80000	0.06423252	0.1178837
4	1.2000	0.05032645	0.1248368
5	1.6000	0.04368964	0.1281552
6	2.0000	0.04029397	0.1298530
7	2.4000	0.03849005	0.1307550
8	2.8000	0.03751184	0.1312441
9	3.2000	0.03697534	0.1315123
10	3.6000	0.03667924	0.1316604
11	4.0000	0.03651526	0.1317424
12	4.4000	0.03642427	0.1317879
13	4.8000	0.03637372	0.1318131
14	5.2000	0.03634563	0.1318272
15	5.6000	0.03633001	0.1318350
16	6.0000	0.03632132	0.1318393
17	6.4000	0.03631649	0.1318418
18	6.8000	0.03631380	0.1318431
19	7.2000	0.03631231	0.1318438
20	7.6000	0.03631148	0.1318443
21	8.0000	0.03631102	0.1318445
22	8.4000	0.03631076	0.1318446
23	8.8000	0.03631062	0.1318447
24	9.2000	0.03631054	0.1318447
25	9.6000	0.03631049	0.1318448
26	10.000	0.03631047	0.1318448

The approach taken here avoids this complication. All of the variables required for program execution are hidden in the supporting software tool, insulated from the user program, so that (providing that the software tool is bug-free) the user of the system need only examine a single, static description of the equations when evaluating the correctness of a program. The BandAid procedure is designed to use the static representation of the equations, exactly as written by the programmer, in the numerical solution of the problem. The procedure in no way modifies the meaning of the description during program execution. In fact, the user programs remain sufficiently independent of the details of the underlying BAND algorithm, that, even if that algorithm were to be changed (to make use of parallel processing, for example), it would not be necessary to rewrite them.

In addition, it is desirable that the functional description of the system of equations should resemble closely the notation used in their original development. Thus, for example,

$$D_A \frac{d^2 c_A}{dx^2} - v\frac{dc_A}{dx} - R_A = 0$$

is written in the DiffEQ function as

```
DiffEQ := DA*d2dx2(cA,x) - v*ddx(cA,x) - RA(x);    { = 0 }
```

This notational similarity is important, because it can be used to make the functional specification self-documenting. That is, the program listing of the differential equations can be made sufficiently similar to the equations themselves that a reader, without the aid of comments, can immediately recognize them. This does not mean that comments will never be necessary or helpful, but it does mean that much of the typical annotation of the program listing can be avoided. The benefit is that, since there are no comments, discrepencies between comments and

code are eliminated[†]. Furthermore, this combining of documentation and code can help not only in the development of new programs, but in the maintenance of existing ones as well. With this approach, when the code is modified, the documentation is simultaneously revised to correspond to the new problem statement.

As demonstrated in the two examples that follow, the program listings for a wide variety of problems can be written to resemble very closely the code for CatReac. Therefore, although the differential equations in the following examples are more complicated than those for the catalytic reactor problem, the basic structure of each computer program is the same, and the complexity of the programs does not increase any faster than the complexity of the problems themselves.

[†] Debugging can be very difficult if comments do not accurately describe the code they claim to represent.

2.5. A Second Example: Flow-Through Porous Electrodes

The second illustrative example also involves a packed-bed reactor. This time, however, the problem is electrochemical, and the kinetic expressions are more complicated. Although the problem is more difficult, the approach to writing the user program is practically identical to that taken in the first example, and, in fact, the program listing for this example was obtained by simple editing of the code for CatReac.

2.5.1. The Mathematical Model

Trainham and Newman [17] have developed the following model, in dimensionless form, for the concentration and potential distribution in a flow-through porous electrode for dilute metal-ion removal of a single species. The model contains two coupled equations: a mass balance,

$$-\frac{dN_R}{dy} = J_R \ , \tag{2-5}$$

and a charge balance,

$$\frac{d^2\eta'}{dy^2} = P_2 (J_R + J_S) \ . \tag{2-6}$$

θ represents metal-ion concentration, η' electric driving force, and y distance through the packed bed. N_R, the reactant-ion flux, is defined as

$$N_R = -D' \frac{d\theta}{dy} + \theta \ , \tag{2-7}$$

J_R, a reaction-rate term for the main reaction (metal deposition), as

$$J_R = \frac{\theta - P_1 \exp\left[\left(\frac{\alpha_{aR}}{\alpha_{cR}} + 1 \right) \eta' \right]}{1 + \exp(\eta')} \ , \tag{2-8}$$

and, J_S, a reaction-rate term for the side reaction (hydrogen evolution), as

$$J_S = P_3 \exp\left(- \frac{\alpha_{cS}}{\alpha_{cR}} \eta' \right) \left[1 - P_4 \exp\left(\frac{\alpha_{aS} + \alpha_{cS}}{\alpha_{cR}} \eta' \right) \right] . \qquad (2\text{-}9)$$

The parameter D' is a dimensionless axial dispersion coefficient. P_1 characterizes the backward term of the main-reaction rate, P_2 the relative importance of ohmic resistance to mass-transfer resistance, P_3 the rate of the side reaction, and P_4 the backward term in the side-reaction rate.

Boundary conditions at the inlet to and exit from the reactor are represented as follows:

at $y = 0$,

$$\theta - D' \frac{d\theta}{dy} = 1 \quad \text{and} \quad \frac{d\eta'}{dy} = -P_2 I^* , \qquad (2\text{-}10)$$

and, at $y = \alpha L$,

$$\frac{d\theta}{dy} = \frac{d\eta'}{dy} = 0 . \qquad (2\text{-}11)$$

The parameter α is the reciprocal of a penetration depth into the reactor, and I^* is the ratio of the current density to the limiting-current density that would exist if all of the metal-ion feed were completely reacted in the absence of a side reaction.

Figure 2-4 shows the user program for the solution of this boundary-value problem with BandAid,† and figure 2-5 shows sample output. (The parameters chosen are those used by Trainham and Newman to model the behavior of a flow-through porous electrode for the removal of copper contamination from sulfuric acid solutions.)

† The independent variable in the Equation function is denoted here by y rather than by x (as in CatReac), so that the notation in the listing corresponds more closely to that used in the mathematical development. The names need not be the same in every program, because all of the parameters to the Equation function are simply formal (dummy) parameters. Similarly, the actual parameters sent to BandAid (such as jMax or even Equation itself!) can be named differently. In fact, only RealNumber, ValueArray, ReadAndPrintParameters, PrintOut, and BandAid are reserved words, since these are the only identifiers declared in a scope global to the user program.

Figure 2-4. Text of Program FlowThru

```
[INHERIT('AidMod pen','FlowIO pen')]

program FlowThru( input, output );

const   n = 2;
        Tolerance = 1e-6;

var     P1, P2, P3, P4, DP, AlphaAR, AlphaCR,
                AlphaAS, AlphaCS, AlphaL, IStar : RealNumber;
        jMax, j, ItMax, It, CPUTime : integer;
        Guess, Result, Residual : ValueArray;


function Equation (
                i, j : integer;

                y, h : RealNumber;

                var NewResult : ValueArray;

                function clnterp( k : integer;
                                  y : RealNumber;
                                  var Result : ValueArray ) : RealNumber;

                function dnFdyn( n : integer;
                                 function F ( y : RealNumber ) : RealNumber;
                                 y : RealNumber                    ) : RealNumber

                                                                   ) : RealNumber;

        function T( y : RealNumber ) : RealNumber;      {... Theta }
        begin   T := clnterp(1,y,NewResult)     end;

        function E( y : RealNumber ) : RealNumber;      {... Eta-Prime }
        begin   E := clnterp(2,y,NewResult)     end;

        function ddy( function G( y : RealNumber ) : RealNumber;
                      y : RealNumber                        ) : RealNumber;
        begin
                ddy := dnFdyn(1,G,y)
        end;

        function d2dy2( function G( y : RealNumber ) : RealNumber;
                        y : RealNumber                        ) : RealNumber;
        begin
                d2dy2 := dnFdyn(2,G,y)
        end;


        function NR( y : RealNumber ) : RealNumber;     {... Reactant flux }
        begin
                NR := -DP*ddy(T,y) + T(y)
        end;


        function JR( y : RealNumber ) : RealNumber;     {... Main-reaction rate }
```

```
        var     Top, Bottom : RealNumber;

        begin
                Top := T(y) - P1*exp( ( (AlphaAR/AlphaCR) + 1 )*E(y) );
                Bottom := 1 + exp( E(y) );

                JR := Top/Bottom
        end;

        function JS( y : RealNumber ) : RealNumber;    {... Side-reaction rate }

        var     exp1, exp2 : RealNumber;

        begin
                exp1 := exp( -(AlphaCS/AlphaCR)*E(y) );
                exp2 := exp( ( (AlphaAS + AlphaCS)/AlphaCR )*E(y) );

                JS := P3*exp1*( 1 - P4*exp2 )
        end;

        function MassBalance( y : RealNumber ) : RealNumber;
        begin
                MassBalance := -ddy(NR,y) - JR(y)    { = 0 }
        end;

        function ChargeBalance( y : RealNumber ) : RealNumber;
        begin
                ChargeBalance := d2dy2(E,y) - P2*( JR(y) + JS(y) )    { = 0 }
        end;

        function UpStrmBC( i : integer ) : RealNumber;
        begin
                case i of

                1:      UpStrmBC := T(0) - DP*ddy(T,0) - 1;  { = 0 }
                2:      UpStrmBC := ddy(E,0) + P2*IStar    { = 0 }

                end  { i cases }
        end;

        function DnStrmBC( i : integer ) : RealNumber;
        begin
                case i of

                1:      DnStrmBC := ddy(T,AlphaL);  { = 0 }
                2:      DnStrmBC := ddy(E,AlphaL)   { = 0 }

                end  { i cases }
        end;

begin  { body of Equation }

        if ( j = 1 ) then
                Equation := UpStrmBC(i)
```

```
        else if ( j > 1 ) and ( j < jMax ) then
                case i of
                1:      Equation := MassBalance(y);
                2:      Equation := ChargeBalance(y)
                end  { i cases }

        else if ( j = jMax ) then
                Equation := DnStrmBC(i)

end;  { Equation }


begin  { body of FlowThru }

        ReadAndPrintParameters( P1, P2, P3, P4, DP, AlphaAR, AlphaCR,
                                AlphaAS, AlphaCS, AlphaL, IStar, jMax, ItMax );

        for j := 1 to jMax do
        begin
                Guess[1,j] := 1;
                Guess[2,j] := 0
        end;

        BandAid( n, jMax, ItMax, It, CPUTime, AlphaL,
                        Tolerance, Equation, Guess, Result, Residual );

        PrintOut( Result, jMax, It, CPUTime, AlphaL )

end.  { FlowThru }
```

Figure 2-5. Sample Output from FlowThru

FlowThru Program

Parameters

P1 = 1.0490E-07
P2 = -3.2540
P3 = 1.2470E-05
P4 = 5.8630E-09
DP = 0.12170
AlphaAR = 1.50
AlphaCR = 0.50
AlphaAS = 0.50
AlphaCS = 0.50

AlphaL = 8.6630
I-Star = 0.95000

jMax = 26
ItMax = 12

Profile Listing

Number of iterations = 6
Execution time = 9310 milli-seconds

j	x	T	E
1	0.00000	0.9087793	-3.326501
2	0.34652	0.6824420	-2.377333
3	0.69304	0.5229007	-1.672223
4	1.0396	0.4103896	-1.139143
5	1.3861	0.3300385	-0.7275479
6	1.7326	0.2714530	-0.4029130
7	2.0791	0.2276855	-0.1418589
8	2.4256	0.1941693	0.07155833
9	2.7722	0.1678955	0.2483940
10	3.1187	0.1468550	0.3964782
11	3.4652	0.1296815	0.5214832
12	3.8117	0.1154263	0.6276105
13	4.1582	0.1034168	0.7180381
14	4.5048	0.09316665	0.7952169
15	4.8513	0.08431748	0.8610707
16	5.1978	0.07660056	0.9171343
17	5.5443	0.06981113	0.9646502
18	5.8908	0.06379091	1.004638
19	6.2374	0.05841585	1.037943
20	6.5839	0.05358760	1.065278
21	6.9304	0.04922702	1.087246
22	7.2769	0.04527167	1.104363
23	7.6234	0.04165978	1.117076
24	7.9700	0.03840698	1.125774
25	8.3165	0.03511203	1.130797
26	8.6630	0.03401371	1.132471

2.6. A Third Example: Flow Generated by a Rotating Disk

As a final introductory example, the solution to the problem of fluid motion to a rotating disk is developed. This particular application is discussed quite thoroughly by White *et al.* [18], where Newman's method is compared to a number of other possible techniques, and interested readers may wish to consult that article for more detail.

2.6.1. Governing Equations

The von Kármán transformation [19] can be used to reduce the equations of motion and continuity in three dimensions (r, θ, and z) to the set of four coupled, nonlinear equations shown below, where $\varsigma = z\sqrt{\Omega/\nu}$:

$$2F + \frac{dH}{d\varsigma} = 0, \tag{2-12}$$

$$F^2 - G^2 + H\frac{dF}{d\varsigma} - \frac{d^2F}{d\varsigma^2} = 0, \tag{2-13}$$

$$2FG + H\frac{dG}{d\varsigma} - \frac{d^2G}{d\varsigma^2} = 0, \tag{2-14}$$

and

$$P = -\frac{1}{2}H^2 - 2F. \tag{2-15}$$

The unknowns are related to the three components of the velocity vector and to the pressure according to the following relations:

$$F = \frac{v_r}{r\Omega}, \tag{2-16}$$

$$G = \frac{v_\theta}{r\Omega}, \tag{2-17}$$

$$H = \frac{v_z}{\sqrt{\nu\Omega}}, \tag{2-18}$$

and

$$P = \frac{\mathrm{P}}{\mu\Omega}, \tag{2-19}$$

and boundary conditions on the velocity variables complete the specification of the problem:

$$F(0) = H(0) = 0,$$

$$G(0) = 1, \tag{2-20}$$

$$F(\infty) = G(\infty) = 0.$$

The system contains a set of three ordinary differential equations (one first-order equation and two second-order equations) and one algebraic equation. Figure 2-6 illustrates how BandAid can be used to construct a solution to this problem,† and figure 2-7 shows sample output from a typical run.

Unlike the first two examples, the system of equations here is defined over a semi-infinite domain ($0 < \varsigma < \infty$). Although the numerical method can only be used on a finite domain, it is expected that the effect of the rotating disk will not be felt far from the surface. Consequently, the program can be run with a number of different values of ς_{max} until changing ς_{max} no longer affects the solution to the problem. (In the run shown here, ς_{max} is 10.) In short, for sufficiently large domains, this solution to the bounded problem should be identical to that for the semi-infinite system. (This point is discussed in detail in [18].) With this approach, the BAND algorithm can be used not only for this problem but for many other boundary-value problems defined on semi-infinite domains as well.

† Since H is one of the components of the velocity vector, the identifier dz (rather than h) represents the mesh size in this program.

Figure 2-6. Text of Program vonKarman

```
|INHERIT('AidMod.pen','KarmanIO.pen')|

program vonKarman( input, output );

const    n = 4;
         Tolerance = 1e-6;
         L = 10;

var      jMax, j, k, ItMax, It, CPUTime : integer;
         Guess, Result, Residual : ValueArray;


function Equation (
                  i, j : integer;

                  z, dz : RealNumber;

                  var NewResult : ValueArray;

                  function cInterp( k : integer;
                                    z : RealNumber;
                                    var Result : ValueArray ) : RealNumber;


                  function dnFdzn( n : integer;
                                   function F ( z : RealNumber ) : RealNumber;
                                   z : RealNumber                    ) : RealNumber

                                                                      ) : RealNumber;


        function F ( z : RealNumber ) : RealNumber;
        begin   F := cInterp(1,z,NewResult)      end;

        function G ( z : RealNumber ) : RealNumber;
        begin   G := cInterp(2,z,NewResult)      end;

        function H( z : RealNumber ) : RealNumber;
        begin   H := cInterp(3,z,NewResult)      end;

        function P( z : RealNumber ) : RealNumber;
        begin   P := cInterp(4,z,NewResult)      end;



        function ddz( function G( z : RealNumber ) : RealNumber;
                          z : RealNumber                ) : RealNumber;
        begin
                ddz := dnFdzn(1,G,z)
        end;


        function d2dz2( function G( z : RealNumber ) : RealNumber;
                            z : RealNumber           ) : RealNumber;
        begin
```

```
            d2dz2 := dnFdzn(2,G,z)
        end;


        function EQ1( z : RealNumber ) : RealNumber;
        begin
            EQ1 := 2*F(z) + ddz(H,z)   { = 0 }
        end;


        function EQ2( z : RealNumber ) : RealNumber;
        begin
            EQ2 := sqr(F(z)) - sqr(G(z)) + H(z)*ddz(F,z) - d2dz2(F,z)   { = 0 }
        end;


        function EQ3( z : RealNumber ) : RealNumber;
        begin
            EQ3 := 2*F(z)*G(z) + H(z)*ddz(G,z) - d2dz2(G,z)   { = 0 }
        end;


        function EQ4( z : RealNumber ) : RealNumber;
        begin
            EQ4 := P(z) + sqr(H(z))/2 - 2*F(z)   { = 0 }
        end;


begin   { body of Equation }

        if ( j = 1 ) then
            case i of

            1:  Equation := H(0);          { = 0 }
            2:  Equation := F(0);          { = 0 }
            3:  Equation := G(0) - 1;      { = 0 }
            4:  Equation := EQ4(0)

            end   { i cases }

        else if ( j > 1 ) and ( j < jMax ) then
            case i of

            1:  Equation := EQ1(z);
            2:  Equation := EQ2(z);
            3:  Equation := EQ3(z);
            4:  Equation := EQ4(z)

            end   { i cases }

        else if ( j = jMax ) then
            case i of

            1:  Equation := EQ1(L);
            2:  Equation := F(L);          { = 0 }
            3:  Equation := G(L);          { = 0 }
            4:  Equation := EQ4(L)

            end   { i cases }
```

```
end;  { Equation }


begin  { body of vonKarman }
          ReadAndPrintParameters( jMax, ItMax );

          for j := 1 to jMax do
               for k := 1 to n do  Guess[k,j] := 0;

          BandAid( n, jMax, ItMax, It, CPUTime, L,
                              Tolerance, Equation, Guess, Result, Residual );

          PrintOut( Result, jMax, It, CPUTime, L );

end  { vonKarman }
```

Figure 2-7. Sample Output from vonKarman

vonKarman Program

Parameters

jMax = 101
ItMax = 12

Profile Listing

Number of iterations = 10
Execution time = 95510 milli-seconds

j	z	F	G	H	P
1	0.00000	0.0000000	1.000000	0.0000000	0.0000000
2	0.100000	0.04615171	0.9384873	-0.005084297	0.09229050
3	0.20000	0.08349592	0.8778719	-0.01846069	0.1668214
4	0.30000	0.1131414	0.8188329	-0.03848266	0.2255424
5	0.40000	0.1361088	0.7618700	-0.06371725	0.2701877
6	0.50000	0.1533290	0.7073362	-0.09292619	0.3023403
7	0.60000	0.1656438	0.6554659	-0.1250488	0.3234689
8	0.70000	0.1738085	0.6063983	-0.1591837	0.3349473

93	9.2000	1.373720E-04	1.798158E-04	-0.8817505	-0.3884673
94	9.3000	1.144586E-04	1.498220E-04	-0.8817756	-0.3885352
95	9.4000	9.348022E-05	1.223615E-04	-0.8817963	-0.3885954
96	9.5000	7.427352E-05	9.722051E-05	-0.8818130	-0.3886485
97	9.6000	5.668893E-05	7.420298E-05	-0.8818260	-0.3886952
98	9.7000	4.058949E-05	5.312956E-05	-0.8818356	-0.3887359
99	9.8000	2.584979E-05	3.383604E-05	-0.8818423	-0.3887712
100	9.9000	1.235501E-05	1.617206E-05	-0.8818460	-0.3888014
101	10.000	0.0000000	0.0000000	-0.8818472	-0.3888272

2.7. Overview of the Remainder of the Presentation

The vonKarman program completes a basic introduction to the programming technique. Most problems solved by the BAND method can be constructed to fit into the form of the programs shown here, and many other problems can be solved by simple modification of the examples. The remainder of this document provides additional information necessary for non-trivial extensions to examples.

The next section contains a description of the BAND algorithm, and a discussion of some of the numerical difficulties associated with its use. This is followed by sections concerned with extensions to the method, particularly for use of the technique to solve a limited class of partial differential equations, and for employment of coordinate transformations to generate non-uniform mesh-point spacings. Finally, the last section of the document is concerned with special features and techniques associated with use of the complete BandShell routine, and the appendix contains a discussion of the implementation of the package on a real computer. It is hoped that these additional details will not obscure the functional style of the problem description found in the examples of this section, since it is the goal of supporting a style similar to that shown in these first three programs that has been the guiding principle in the development of BandAid.

2.8. The BAND Algorithm

The basis for all of the calculations performed by BandAid is the BAND algorithm developed by John Newman. This technique is a banded-matrix method for solving the sets of coupled, nonlinear algebraic equations often obtained in the finite-difference representation of elliptic boundary-value problems. A (somewhat liberal) translation of Newman's original FORTRAN code has been used to construct the majority of the BandCore procedure, which forms the heart of BandShell. Thus, a description of the technique is given here so that the foundations of the BandAid program may be examined. For a more complete discussion, readers unfamiliar with the method should refer to the original articles by Newman [1,2,3] and to the more detailed explanation by White *et al.* [18].

2.8.1. Interior Mesh Points

In general, any differential equation may be represented, at an interior mesh point "j" of a discretized domain, by a finite-difference expression involving only the values of the dependent variables at that interior point (j), the point before it ($j-1$), and the point after it ($j+1$). Thus, for example, the coupled set of equations for the catalytic reactor problem

$$D_A \frac{d^2 c_A}{dx^2} - v\frac{dc_A}{dx} - (k_f c_A^2 - k_b c_B) = 0 , \qquad (2\text{-}21)$$

and

$$D_B \frac{d^2 c_B}{dx^2} - v\frac{dc_B}{dx} + \frac{1}{2}(k_f c_A^2 - k_b c_B) = 0 , \qquad (2\text{-}22)$$

can be written in finite-difference form at a point j (corresponding to a distance x_j in the domain) as

$$D_A \frac{c_{1,j+1} + c_{1,j-1} - 2c_{1,j}}{h^2} - v\frac{c_{1,j+1} - c_{1,j-1}}{2h} - (k_f c_{1,j}^2 - k_b c_{2,j}) = 0 , \qquad (2\text{-}23)$$

and

$$D_B \frac{c_{2,j+1} + c_{2,j-1} - 2c_{2,j}}{h^2} - v\frac{c_{2,j+1} - c_{2,j-1}}{2h} + \frac{1}{2}(k_f c_{1,j}^2 - k_b c_{2,j}) = 0 , \quad (2\text{-}24)$$

where

h is the mesh size,

$c_{1,j-1}$ and $c_{2,j-1}$ are the values of A and B, respectively, at $x_j - h$,

$c_{1,j}$ and $c_{2,j}$ are the values at x_j,

and

$c_{1,j+1}$ and $c_{2,j+1}$ are the values at $x_j + h$.

In the representation shown above, all of the derivatives are approximated by central-difference formulas, which are accurate to order h^2. Other finite-difference approximations are sometimes used, particularly at the boundaries of the domain, and these are shown in the section on boundary points that follows.

In general, then, the i^{th} equation of a set of n (i. e., n equations with n unknowns) may be represented, at a point j, as

$$F_i(x_j , c_{1,j-1} , c_{2,j-1} , \cdots , c_{n,j-1} , c_{1,j} , \cdots , c_{n,j+1}) = 0 , \quad (2\text{-}25)$$

where, in this example,

$$F_1 = D_A \frac{c_{1,j+1} + c_{1,j-1} - 2c_{1,j}}{h^2} - v\frac{c_{1,j+1} - c_{1,j-1}}{2h} - (k_f c_{1,j}^2 - k_b c_{2,j}) \quad (2\text{-}26)$$

and

$$F_2 = D_B \frac{c_{2,j+1} + c_{2,j-1} - 2c_{2,j}}{h^2} - v\frac{c_{2,j+1} - c_{2,j-1}}{2h} + \frac{1}{2}(k_f c_{1,j}^2 - k_b c_{2,j}) . \quad (2\text{-}27)$$

This completes the discretization of the problem, but one further step must still be taken before the discretized equations can be solved. The BAND algorithm is intended to be used in the solution of sets of coupled, *linear*, difference equations. If the equations are *nonlinear*, as they are here, the method must be combined with an iterative procedure that solves linear problems (i. e., sets of equations linearized about a trial solution) at each step. The trial solutions are updated

after each iteration until convergence of the nonlinear problem is achieved.

An expression F_i can be linearized about a trial solution by construction of a Taylor series expansion, as follows:

$$F_i = F_i^{\circ} + \left.\frac{\partial F_i}{\partial c_{1,j-1}}\right|_{\circ} \Delta c_{1,j-1} + \left.\frac{\partial F_i}{\partial c_{2,j-1}}\right|_{\circ} \Delta c_{2,j-1} + \cdots + \left.\frac{\partial F_i}{\partial c_{n,j-1}}\right|_{\circ} \Delta c_{n,j-1} +$$

$$\left.\frac{\partial F_i}{\partial c_{1,j}}\right|_{\circ} \Delta c_{1,j} + \cdots + \left.\frac{\partial F_i}{\partial c_{n,j+1}}\right|_{\circ} \Delta c_{n,j+1} , \qquad (2\text{-}28)$$

where terms of quadratic and higher order have been neglected.

$\left.\frac{\partial F_i}{\partial c_{k,j}}\right|_{\circ}$ represents the derivative of the expression F_i with respect to $c_{k,j}$ at the trial conditions ($^{\circ}$), and $\Delta c_{k,j}$ is the difference between the value of the k^{th} variable at a point j and its trial value at that point, i. e.,

$$\Delta c_{k,j} = c_{k,j} - c_{k,j}^{\circ} . \qquad (2\text{-}29)$$

The function F_i may now be replaced by its linearized form,

$$\bar{F}_i(x_j , \Delta c_{1,j-1} , \Delta c_{2,j-1} , \cdots , \Delta c_{n,j-1} , \Delta c_{1,j} , \cdots , \Delta c_{n,j+1}) = 0 , \qquad (2\text{-}30)$$

and rewritten as

$$A_{i,1}\Delta c_{1,j-1} + A_{i,2}\Delta c_{2,j-1} + \cdots + A_{i,n}\Delta c_{n,j-1} + B_{i,1}\Delta c_{1,j} + \cdots$$

$$+ D_{i,n}\Delta c_{n,j+1} - G_i = 0 , \qquad (2\text{-}31)$$

which represents the i^{th} equation in a set of n, coupled, *linear*, difference equations. Here, $-G_i$ at any point is simply the trial value of the function, F_i°, and the $A_{i,k}$, $B_{i,k}$, and $D_{i,k}$ represent the appropriate derivatives of F_i at the trial conditions. That is,

$$A_{i,k,j} = \left.\frac{\partial F_i}{\partial c_{k,j-1}}\right|_o , \tag{2-32}$$

$$B_{i,k,j} = \left.\frac{\partial F_i}{\partial c_{k,j}}\right|_o , \tag{2-33}$$

$$D_{i,k,j} = \left.\frac{\partial F_i}{\partial c_{k,j+1}}\right|_o , \tag{2-34}$$

and

$$G_{i,j} = -F_i^{\circ} . \tag{2-35}$$

Thus, for example, in the catalytic-reactor equations shown above

$$\begin{aligned} A_{1,1,j} &= \frac{D_A}{h^2} + \frac{v}{2h} , & A_{1,2,j} &= 0 , \\ A_{2,1,j} &= 0 , & A_{2,2,j} &= \frac{D_B}{h^2} + \frac{v}{2h} , \end{aligned}$$

$$\begin{aligned} B_{1,1,j} &= \frac{-2D_A}{h^2} - 2k_f\, c^{\circ}_{1,j} , & B_{1,2,j} &= k_b , \\ B_{2,1,j} &= k_f\, c^{\circ}_{1,j} , & B_{2,2,j} &= \frac{-2D_B}{h^2} - \frac{k_b}{2} , \end{aligned} \tag{2-36}$$

$$\begin{aligned} D_{1,1,j} &= \frac{D_A}{h^2} - \frac{v}{2h} , & D_{1,2,j} &= 0 , \\ D_{2,1,j} &= 0 , & D_{2,2,j} &= \frac{D_B}{h^2} - \frac{v}{2h} , \end{aligned}$$

$$G_{1,j} = -D_A \frac{c^{\circ}_{1,j+1} + c^{\circ}_{1,j-1} - 2c^{\circ}_{1,j}}{h^2} + v\frac{c^{\circ}_{1,j+1} - c^{\circ}_{1,j-1}}{2h} + \left(k_f \left(c^{\circ}_{1,j}\right)^2 - k_b\, c^{\circ}_{2,j} \right),$$

and,

$$G_{2,j} = -D_B \frac{c^{\circ}_{2,j+1} + c^{\circ}_{2,j-1} - 2c^{\circ}_{2,j}}{h^2} + v\frac{c^{\circ}_{2,j+1} - c^{\circ}_{2,j-1}}{2h} - \frac{1}{2}\left(k_f \left(c^{\circ}_{1,j}\right)^2 - k_b\, c^{\circ}_{2,j} \right).$$

The complete set of equations may be written in matrix form as follows:

$$\begin{bmatrix} |B_{1,k,1}| & |D_{1,k,1}| & & & & \\ |A_{1,k,2}| & |B_{1,k,2}| & |D_{1,k,2}| & & & \\ & \ddots & \ddots & \ddots & & \\ & & & |A_{1,k,j_{max}-1}| & |B_{1,k,j_{max}-1}| & |D_{1,k,j_{max}-1}| \\ & & & & |A_{1,k,j_{max}}| & |B_{1,k,j_{max}}| \end{bmatrix} \begin{bmatrix} |\Delta c_{k,1}| \\ |\Delta c_{k,2}| \\ \vdots \\ |\Delta c_{k,j_{max}-1}| \\ |\Delta c_{k,j_{max}}| \end{bmatrix} = \begin{bmatrix} |G_{1,1}| \\ |G_{1,2}| \\ \vdots \\ |G_{1,j_{max}-1}| \\ |G_{1,j_{max}}| \end{bmatrix} \tag{2-37}$$

2.8.2. Boundary Points

Boundary conditions need to be treated in a slightly different manner from the equations in the interior. Notice that the matrix representation of the set of equations, although completely tridiagonal in the interior, contains only two entries in the upper-left and lower-right corners. A central-difference approximation to a derivative on the boundary of the domain cannot be used directly, since it requires the value of the dependent variable outside of the domain (i. e., $j = 0$ or $j = j_{max}+1$). One method for dealing with this difficulty is to avoid altogether the use of central-difference approximations at the boundaries, and this is the approach taken in BandAid. With this method, a three-point forward difference approximation (accurate to order h^2, as is a

central difference) is employed at the left boundary ($j = 1$). Thus, at $j = 1$,

$$\frac{dc_k}{dx} = \frac{-3c_{k,j} + 4c_{k,j+1} - c_{k,j+2}}{2h} . \tag{2-38}$$

Similarly, at the right boundary, an analogous expression (three-point backward difference) is employed. Thus, at $j = j_{max}$,

$$\frac{dc_k}{dx} = \frac{3c_{k,j} - 4c_{k,j-1} + c_{k,j-2}}{2h} . \tag{2-39}$$

Although this approach avoids the problems of points outside the domain, off-tridiagonal elements $(j+2)$ and $(j-2)$ are still introduced at the top and bottom of the coefficient matrix. Newman's algorithm, however, provides for this possibility, and it may be used to solve the problem directly. Thus, the complete set of equations and boundary conditions may be written as

$$\begin{bmatrix} |B_{i,k,1}| & |D_{i,k,1}| & |X_{i,k,1}| & & & \\ |A_{i,k,2}| & |B_{i,k,2}| & |D_{i,k,2}| & & & \\ & \ddots & & \ddots & & \\ & & & |A_{i,k,j_{max}-1}| & |B_{i,k,j_{max}-1}| & |D_{i,k,j_{max}-1}| \\ & & & |Y_{i,k,j_{max}}| & |A_{i,k,j_{max}}| & |B_{i,k,j_{max}}| \end{bmatrix} \cdot \begin{bmatrix} |\Delta c_{k,1}| \\ |\Delta c_{k,2}| \\ \vdots \\ |\Delta c_{k,j_{max}-1}| \\ |\Delta c_{k,j_{max}}| \end{bmatrix} = \begin{bmatrix} |G_{i,1}| \\ |G_{i,2}| \\ \vdots \\ |G_{i,j_{max}-1}| \\ |G_{i,j_{max}}| \end{bmatrix} , \tag{2-40}$$

where,

$$X_{i,k,j} = \begin{cases} \left.\dfrac{\partial F_i}{\partial c_{k,j+2}}\right|_o & \text{for } j = 1 \\ 0 \text{ for } j \neq 1 \end{cases} , \tag{2-41}$$

and,

$$Y_{i,k,j} = \begin{cases} \left.\dfrac{\partial F_i}{\partial c_{k,j-2}}\right|_o & \text{for } j = j_{\max} \\ 0 \text{ for } j \neq j_{\max} \end{cases} \tag{2-42}$$

An alternative construction, employing image points and retaining central-difference approximations at the boundaries, can be used instead of the three-point formulas, and this alternative is available when using the complete BandShell procedure. With the image-point approach, the structure of the resulting matrix equation remains the same, and, therefore, the BAND algorithm can also be used directly in that case. Consequently, for most applications the decision to use image points is largely a matter of convenience.

2.8.3. Solving the Matrix Equations

At each iteration with a new trial solution, the coefficient matrix must be inverted so that the deviations $\Delta c_{k,j}$ may be determined. This coefficient matrix ($j_{\max} \times j_{\max}$), itself composed of $n \times n$ matrices, is block-tridiagonal, and a generalization of the Thomas algorithm for the solution of block-tridiagonal matrices containing single elements may be used. The method involves the transformation of the coefficient matrix and the G vector into an upper triangular matrix and a modified vector as shown below:

$$\begin{bmatrix} 1 & |E_{k,1,1}| & & & & \\ & 1 & |E_{k,1,2}| & & & \\ & & \ddots & \ddots & & \\ & & & 1 & |E_{k,1,j_{\max}-1}| & \\ & & & & 1 & \end{bmatrix} \cdot \begin{bmatrix} |\Delta c_{k,1}| \\ |\Delta c_{k,2}| \\ \vdots \\ |\Delta c_{k,j_{\max}-1}| \\ |\Delta c_{k,j_{\max}}| \end{bmatrix} = \begin{bmatrix} |\xi_{k,1}| \\ |\xi_{k,2}| \\ \vdots \\ |\xi_{k,j_{\max}-1}| \\ |\xi_{k,j_{\max}}| \end{bmatrix} \tag{2-43}$$

At each point j, this transformation requires a matrix $[E_{k,j}]$ and a vector $[\xi_k]$ to be determined.

and the necessary calculations are accomplished in the BAND algorithm, one row at a time (starting at $j = 1$), according to the transformation equations found in [1]. The $[E_{k,i,j}]$ and $[\xi_{k,j}]$ are stored, and the $\Delta c_{k,j}$ are then determined by a simple back-substitution procedure (starting at $j = j_{max}$). Thus, the total process involves two passes through the mesh positions in opposite directions. The method is direct and exact in a single operation. As a result, no iteration is required to solve the linear equations, and the only iteration is the outer loop about the trial solutions. (This may be contrasted with predictor-corrector schemes, which require iteration for the linear boundary-value problem.) Once the $\Delta c_{k,j}$ have been determined, the $c^{o}_{k,j}$ are updated, and the process is repeated until convergence is achieved. (It should be noted that earlier applications of the BAND method, such as that described by White *et al.* [18], used the algorithm to solve directly for the $c_{k,j}$ rather than for the $\Delta c_{k,j}$. Although the basic principles are the same, the earlier method requires some additional algebraic manipulation.)

Figure 2-8 shows an outline of the resulting BandCore algorithm. The method depends primarily upon calculation of the derivatives from the finite-difference representation of the differential equations, *i. e.*,

$$\left.\frac{\partial F_i}{\partial c_{k,j}}\right|_{o},$$

and the original BAND subroutine [3] required that the main calling program contain these derivatives, which were usually calculated analytically by the programmer. The principle of the current approach is that the process can be easily and efficiently automated with machine calculation of the derivatives by numerical differentiation. In this way, all possible equations may be solved by the same algorithm, provided that a convenient method is available for the introduction of the equations themselves. By freeing the programmer from much time-consuming algebraic manipulation, the automatic determination of derivatives should make the development of new programs considerably easier, and, in fact, the utility of the approach for shortening the time needed to write and debug programs has already been demonstrated [20].

Figure 2-8. An Outline of the BandCore Algorithm

---

begin

$[c^{\circ}_{k,j}] := [c^{guess}_{k,j}]$;

repeat

for $j := 1$ **to** j_{max} **do**

begin

calculate $A_{i,k,j}$, $B_{i,k,j}$, $D_{i,k,j}$, $X_{i,k,j}$, $Y_{i,k,j}$, and $G_{i,j}$ from F°_i and $\left.\frac{\partial F_i}{\partial c_{k,j}}\right|_{\circ}$;

from $A_{i,k,j}$, $B_{i,k,j}$, $D_{i,k,j}$, $X_{i,k,j}$, $Y_{i,k,j}$, and $G_{i,j}$, calculate $E_{k,i,j}$ and $\xi_{k,j}$ according to the transformation equations

end;

for $j := j_{max}$ **down to** 1 **do**

begin

from $E_{k,i,j}$ and $\xi_{k,j}$, calculate $\Delta c_{k,j}$ according to the back-substitution equations

end;

from the old $[c^{\circ}_{k,j}]$ and the new $[\Delta c_{k,j}]$, calculate the new $[c^{\circ}_{k,j}]$

until convergence is achieved

end

The BandShell procedure extends this process by providing an interface so that the user can write the equations over a continuous domain in a form resembling very closely the original statement of the problem. The procedure, then, performs the discretization of these equations and the differentiation needed to calculate the A, B, D, X, and Y matrices, as well as the mechanics of the BAND algorithm as described above.

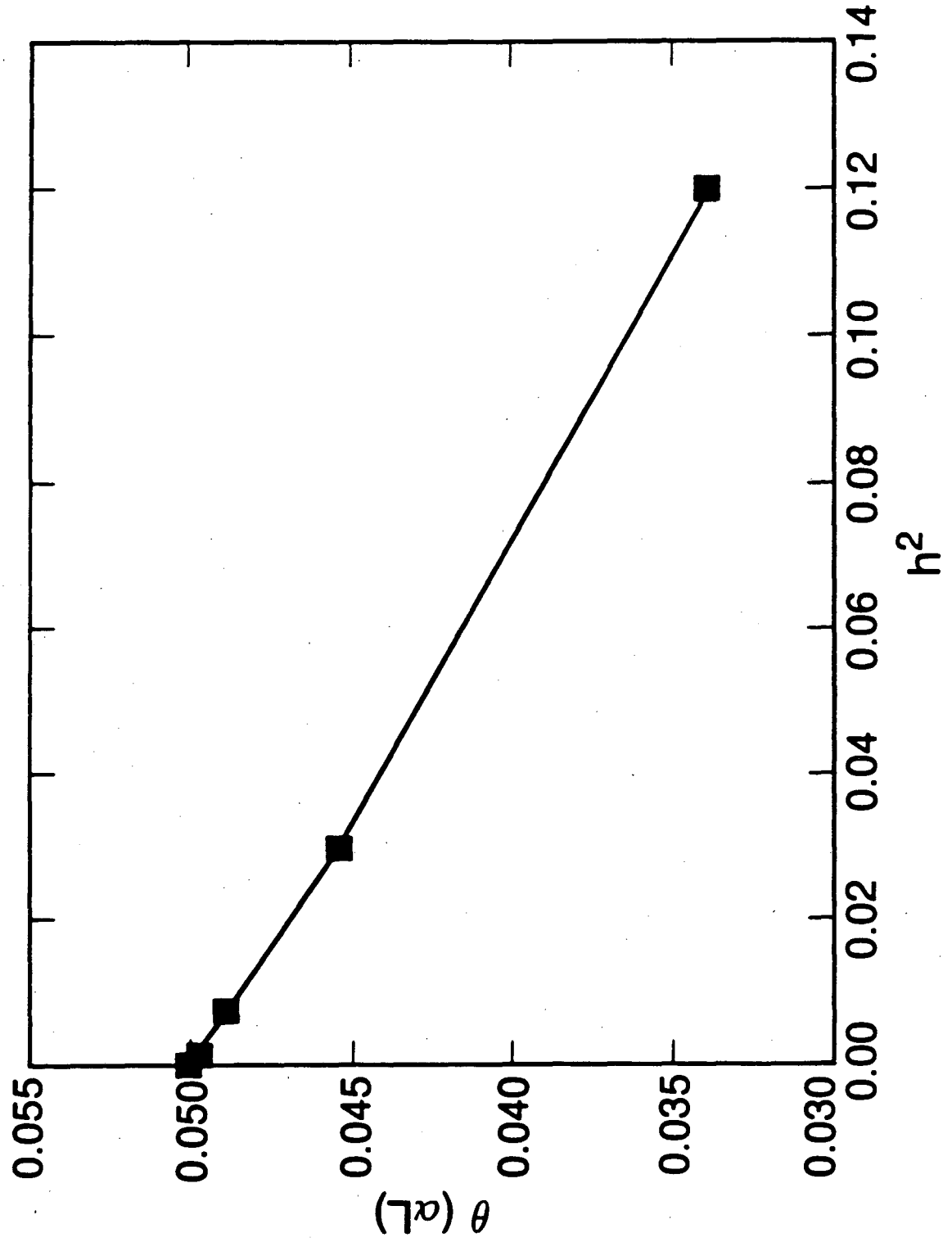
2.8.4. Accuracy of the Method

If all of the finite-difference approximations for the numerical method are taken, as they are here, to be accurate to order h^2, the resulting $c_{k,j}$ should be accurate to order h^2. Consequently, a direct check on the correctness of a converged solution can be made by plotting c_k values obtained at a number of mesh sizes h versus the square of the mesh size, h^2. For very large values of h, such a plot will not be linear owing to the existence of inaccuracies of order h^3 and higher. These higher-order inaccuracies should become insignificant at smaller h, however, and a very linear plot of c_k versus h^2 should result. In fact, such plots can be used to extrapolate to the solution that would be obtained with an infinite number of mesh points (*i. e.*, $h^2 = 0$), and this is often done when results of high accuracy are needed.

Table 2-1. Effect of h^2 on $\theta(\alpha L)$

j_{max}	h	h^2	$\theta(\alpha L)$
26	0.34652	0.12006225	0.03401371
51	0.17326	0.03003289	0.04540695
101	0.08663	0.00750476	0.04887463
201	0.04332	0.00187619	0.04981465
401	0.02166	0.00046905	0.05005691

To illustrate this feature of the solution, table 2-1 contains values of the effluent concentration, $\theta(\alpha L)$, from the flow-through porous electrode reactor of the second example. Each value is taken from a run of the program with a different value of the mesh size. Figure 2-9 shows a plot of the values from the table, indicating that the result is indeed linear in h^2. Table 2-1 also illustrates a general rule of thumb for these systems, namely, that the accuracy of a result increases roughly by one decimal place for every doubling of the mesh size.



XBL 852-8172

Figure 2-9. Plot of $\theta(\alpha L)$ vs. h^2 from table 2-1.

2.8.5. Quadratic Convergence Behavior

Another feature of the iterative scheme outlined in figure 2-8 is its quadratic convergence behavior. As the number of iterations N increases, the error associated with each new iteration becomes proportional to the square of the error at the preceding iteration. Thus, the number of significant digits is doubled at each successive iteration until the limits of floating-point arithmetic on the machine are reached. (A technique that has this behavior is also referred to as a second-order method.)

Table 2-2. Convergence of $\theta(\alpha L)$

N	$\theta_N(\alpha L)$	$\theta_N(\alpha L) - \theta_{N-1}(\alpha L)$
0	1.000000	–
1	0.02741381	-9.725862×10^{-1}
2	0.03341484	6.001022×10^{-3}
3	0.03401037	5.955363×10^{-4}
4	0.03401310	2.732442×10^{-6}
5	0.03401371	6.039986×10^{-7}
6	0.03401371	-1.619286×10^{-11}
7	0.03401371	-1.647987×10^{-16}
8	0.03401371	1.327063×10^{-16}
9	0.03401371	-1.561251×10^{-17}
10	0.03401371	-5.898056×10^{-17}
11	0.03401371	-4.336809×10^{-17}

Table 2-2 shows the convergence behavior for the effluent concentration from the flow-through porous electrode program. Initially, the convergence is relatively slow, since the guess is very far from the correct answer. Once the result is approached (iterations 5, 6, and 7), however, the convergence becomes quadratic, and the error is reduced quickly to the minimum of machine accuracy (1×10^{-16}). The convergence speed is shown in figure 2-10, a semi-logarithmic plot of error ($|\theta_N - \theta_{N-1}|$) versus iteration (N).

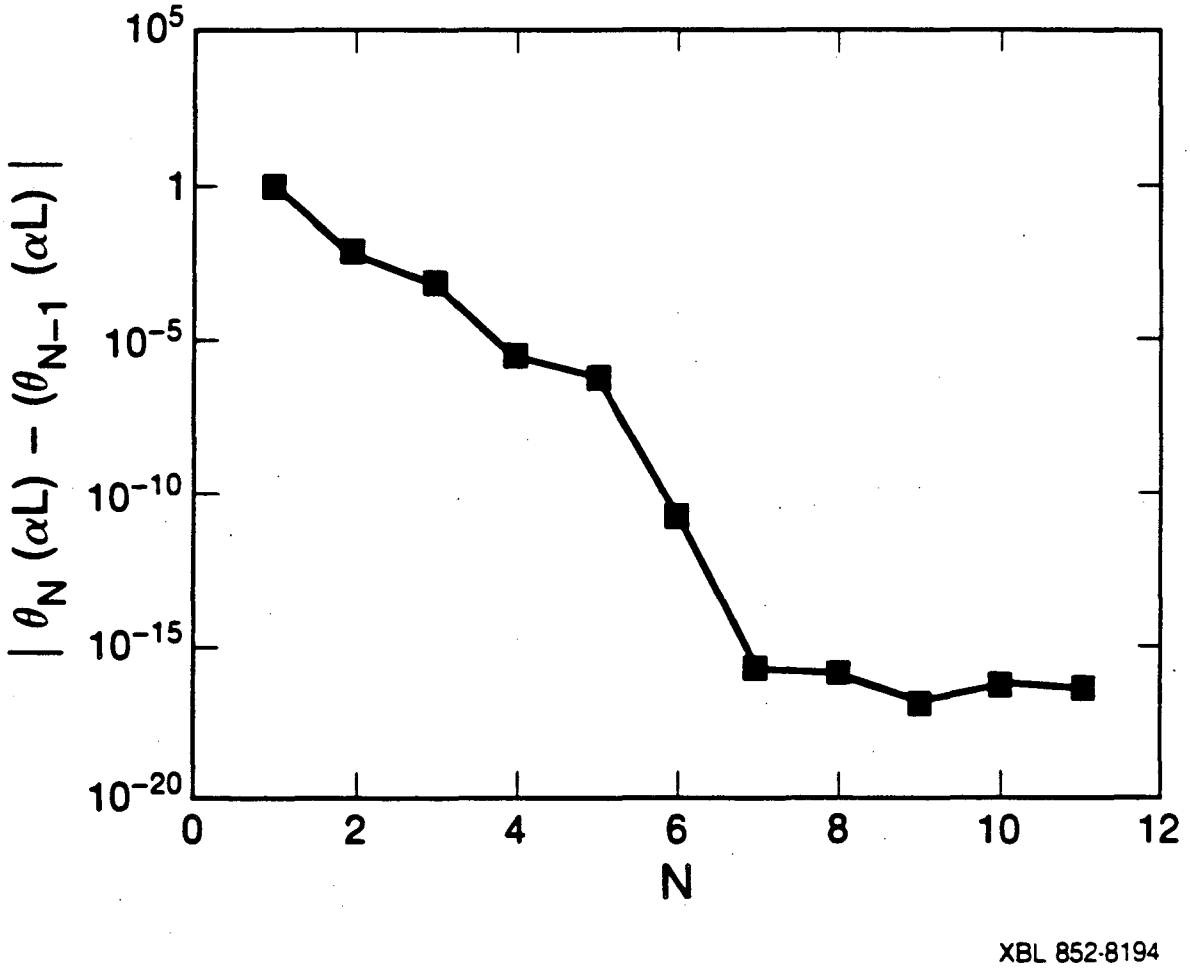


Figure 2-10. Plot of the convergence behavior for the calculation of effluent concentration from the flow-through porous electrode.

2.8.6. Numerical Differentiation

A forward-difference formula is used for the numerical differentiation, i. e.,

$$\frac{dF}{dc_{k,j}} = \frac{F(c_{k,j}+\epsilon) - F(c_{k,j})}{\epsilon} \quad . \tag{2-44}$$

Because $F(c_{k,j})$ can be stored, this formulation requires only a single function call for $F(c_{k,j}+\epsilon)$ for each coefficient calculation, rather than the two calls that would be necessary for a central-difference. In addition, a forward-difference formula avoids the potentially dangerous need for $c_{k,j} - \epsilon$ in cases where $c_{k,j}$ represents concentration and its value is near zero. Experience has shown that this formula is sufficient for all problems of practical interest, and, therefore, BandShell does not provide an alternative.

Numerical inaccuracies in these derivatives represent an additional source of error in the BandShell approach that does not exist in the original BAND algorithm. Since the coefficient matrix is determined from these calculations, inaccuracies in the derivatives may result in some degradation of the performance of the algorithm. Large errors will in all likelihood cause the routine to fail to converge upon a result. It is anticipated, however, that slight errors in the derivatives are not necessarily damaging to the accuracy of the result if, in fact, the routine does succeed in determining the result. This is possible, since the method solves only for the deviations, $\Delta c_{k,j}$, of the $c_{k,j}$ values from their values at the previous iteration, and, in the Taylor series expansion, the derivatives are multiplied by these deviations. Thus, when the routine converges upon the correct result, even if the derivatives are slightly in error, they may not create a large problem, since they are multiplying small $\Delta c_{k,j}$ terms. Errors in the derivatives will, however, affect the convergence properties, resulting in a shift from second-order to first-order behavior as the answer is approached.

In order to avoid the problem of severe roundoff error, it is important that the increment ϵ used in the differentiation formula be chosen carefully. Consequently, the BandShell procedure uses a scaling process to determine the value of ϵ:

$$\epsilon = \text{FactorIncrement} \cdot | c_{k,j} | . \tag{2-45}$$

In BandAid, the FactorIncrement is set to 1×10^{-6}, but this is not a complete specification, since an ϵ of zero is unacceptable. Therefore, if $| c_{k,j} |$ is very small, an alternative is employed:

$$\epsilon = \text{AbsoluteIncrement} . \tag{2-46}$$

Since the dependent variables may not be of order 1, the AbsoluteIncrement is generally different from the FactorIncrement. In BandAid they are both set to the same value, but with BandShell, as shown later, FactorIncrement and AbsoluteIncrement may be set by the user program.

A possible pitfall should be noted here. If the function being evaluated consists of a large constant added to a function of $c_{k,j}$, such as, for example,

$$F_i = c_{k,j} + 1\times10^{25} , \tag{2-47}$$

it is important that the initial guess of $c_{k,j}$ be of an appropriate order of magnitude (here, 1×10^{25}). If the initial guess of $c_{k,j}$ is of incorrect order (say, 1×10^{-5}), it will be practically impossible (on a conventional computer) for the numerical differentiation routine to pick out the $c_{k,j}$ from the large constant. This is simply due to the finite accuracy of floating-point arithmetic, and it is a hazard inherent in numerical differentiation. Certainly, proper scaling and nondimensionalization of the equations are always recommended.

2.8.7. The BandShell Interface: cInterp and dnFdxn

The interpolation operator cInterp creates, by linear interpolation, a continuous function from a set of discrete points. Thus, if x is located between two points x_{j_1} and x_{j_2}, $c_1(x)$ is determined from a relationship such as

$$c_1(x) = c_{1,j_1} + \frac{c_{1,j_1} - c_{1,j_2}}{x_{j_1} - x_{j_2}} \cdot (x - x_{j_1}) , \tag{2-48}$$

where special care is taken to avoid problems with roundoff error if x is very close to x_{j_1} or x_{j_2}. (See the BandShell source code for the exact algorithm.)

The function dnFdxn operates on continuous functions, most of which come from cInterp directly or through composites. For a central-difference approximation, the first derivative of a function $F(x)$ is calculated by dnFdxn as

$$\frac{dF}{dx}(x) = \frac{F(x + h/2) - F(x - h/2)}{h} . \tag{2-49}$$

Notice that this approximation uses half-point values. When $F(x)$ is a simple redefinition from cInterp, it can be shown that this representation is equivalent to the central-difference form shown in the preceding sections on the BAND algorithm. In this case, the first derivative is defined as

$$\frac{dc_k}{dx}(x) = \frac{c_k(x + h/2) - c_k(x - h/2)}{h} . \tag{2-50}$$

Since, by linear interpolation,

$$c_k(x + h/2) = \frac{c_k(x + h) + c_k(x)}{2} , \tag{2-51}$$

and

$$c_k(x - h/2) = \frac{c_k(x) + c_k(x - h)}{2} , \tag{2-52}$$

this expression for the derivative is equivalent to

$$\frac{dc_k}{dx}(x) = \frac{c_k(x + h) - c_k(x - h)}{2h}, \tag{2-53}$$

which is the same as the traditional formula used in the development earlier. With the same definitions, then, a second derivative operator consistent with the traditional form can be defined by recursive application of the first-derivative operator.

If the function of the dependent variables is a nonlinear composite, however, it is important to recognize that the recursive application of dnFdxn, which is a linear difference approximation, may not lead to the desired result. In that event, the user may wish to combine some analytical differentiation with the use of dnFdxn. An example of this approach is shown below.

Consider, for example, the solution to the simple differential material-balance equation:

$$-\frac{dN}{dx} = 0, \tag{2-54}$$

where

$$N = -D\frac{dc}{dx}. \tag{2-55}$$

A program fragment from an Equation function describing this problem can be written as follows, where the first-derivative operator and definition of a composite function are used:

```
function DiffEQ ( x : RealNumber ) : RealNumber;

        function c( x : RealNumber ) : RealNumber;
        begin   c := cInterp(1,x,NewResult)       end;

        function ddx( function G( x : RealNumber );
                      x : RealNumber             ) : RealNumber;
        begin
              ddx := dnFdxn(1,G,x)
        end;

        function N ( x : RealNumber ) : RealNumber;
        begin   N := -D*ddx(c,x)        end;

begin
```

```
        DiffEQ := -ddx(N,x)     { = 0 }
end;
```

Another possibility, however, is to use the second derivative operator as in the example that follows:

```
function DiffEQ ( x : RealNumber ) : RealNumber;

        function c( x : RealNumber ) : RealNumber;
        begin   c := cInterp(1,x,NewResult)       end;

        function d2dx2( function G( x : RealNumber);
                        x : RealNumber                    ) : RealNumber;
        begin
                d2dx2 := dnFdxn(2,G,x)
        end;

begin
        DiffEQ := D*d2dx2(c,x) { = 0 }
end;
```

These two program fragments are equivalent, since the function N(x) is linear in c(x). If the diffusion coefficient is not constant, however, but rather is a function of concentration, the two approaches may not be equivalent. For example, if the diffusion coefficient has a quadratic concentration dependence

$$D = D_o + ac^2, \tag{2-56}$$

where D_o is the value of the diffusion coefficient at infinite dilution, and a is a proportionality constant, two approaches to the program fragment may be determined as follows:

```
function DiffEQ ( x : RealNumber ) : RealNumber;

        function c( x : RealNumber ) : RealNumber;
        begin   c := cInterp(1,x,NewResult)       end;


        function D( c : RealNumber ) : RealNumber;
        begin   D := D0 + a*sqr(c)     end;


        function ddx( function G( x : RealNumber);
                        x : RealNumber                  ) : RealNumber;
        begin
                ddx := dnFdxn(1,G,x)
        end;


        function N( x : RealNumber ) : RealNumber;
        begin   N := -D(c(x))*ddx(c,x)              end;


begin

        DiffEQ := -ddx(N,x)     { = 0 }

end;
```

or,

```
function DiffEQ ( x : RealNumber ) : RealNumber;

        function c( x : RealNumber ) : RealNumber;
        begin   c := cInterp(1,x,NewResult)       end;


        function ddx( function G( x : RealNumber);
                        x : RealNumber                  ) : RealNumber;
        begin
                ddx := dnFdxn(1,G,x)
        end;


        function d2dx2( function G( x : RealNumber);
                        x : RealNumber                  ) : RealNumber;
        begin
                d2dx2 := dnFdxn(2,G,x)
        end;


begin
```

```
DiffEQ := a*( 2*c(x)*ddx(c,x) + sqr(c(x))*d2dx2(c,x) )    { = 0 }
end.
```

In the first case, the finite-difference representation of the differential equation is

$$\frac{a\left(\frac{c_j + c_{j+1}}{2}\right)^2 \frac{c_{j+1} - c_j}{h} - a\left(\frac{c_{j-1} + c_j}{2}\right)^2 \frac{c_j - c_{j-1}}{h}}{h} = 0 , \tag{2-57}$$

whereas, in the second case,

$$2ac_j \left(\frac{c_{j+1} - c_{j-1}}{2h}\right)^2 + ac_j{}^2 \frac{c_{j+1} + c_{j-1} - 2c_j}{h^2} = 0 . \tag{2-58}$$

Although each of these approaches yields the same c_j from an extrapolation to $h^2 = 0$, they are not equivalent representations and produce different results for finite mesh sizes. It is important, therefore, to consider the nature of the representation when specifying the equations, and it is one of the benefits of the programming style advocated here that these decisions concerning the numerical representation can be made explicit and can be documented clearly in the program listing.

2.8.8. Special Considerations for Undetermined Constants and First Derivatives

Perhaps the most common difficulty encountered by new users of the BAND algorithm is the presence of oscillatory results, or, sometimes, complete failure of the BAND algorithm, when attempting to solve systems of equations containing first derivatives (initial-value problems) or undetermined constants. The difficulty is that the central-difference approximation to the first derivative

$$\frac{dc_k}{dx} = \frac{c_{k,j+1} - c_{k,j-1}}{2h} \tag{2-59}$$

contains no contribution from $c_{k,j}$, but, rather, only from $c_{k,j-1}$ and $c_{k,j+1}$. Oscillations arise because information cannot be passed to adjacent points in the domain, and this results in two solutions, one on the even-numbered points and another on the odd-numbered points. It often happens, therefore, that an otherwise well-posed boundary-value problem must be rearranged in order to avoid these difficulties. (And, generally, it is the simplest problems that cause the most trouble.) There is no problem, however, if all of the equations contain second-derivatives or are simple algebraic equations, since in that case a term containing $c_{k,j}$ always appears.

A related problem is that the BAND algorithm, in order to determine the decomposition matrix $[E_{k,i}]$, must invert a matrix very closely related to the matrix $[B_{i,k}]$, and, consequently, $[B_{i,k}]$ matrices having a zero determinant may prevent the routine from determining a result. (When BandAid is used, an error message is produced indicating that a zero-determinant has been encountered, the matrix entries are printed on the standard output, and program execution is stopped.)

It can be seen why first-derivatives produce this problem. For example, if

$$F_i = \frac{dc_k}{dx} \tag{2-60}$$

then, since

$$B_{i,k,j} = \left. \frac{\partial F_i}{\partial c_{k,j}} \right|_o , \tag{2-61}$$

$B_{i,k,j} = 0$ and a zero determinant may result.

These same difficulties arise in the determination of undetermined constants. Edwards and Newman [21], for example, used the BAND algorithm to calculate the eigenvalues and eigenfunctions for the following equation, which arises from a separation of variables solution to the asymmetric Graetz problem:

$$\frac{d^2R}{dx^2} + \lambda^2 (1 - x^2) R = 0 ,$$

where λ is a constant to be determined. Using the fact that the undetermined constant does not change from point to point, they created the following set of two coupled differential equations

$$\frac{d^2R}{dx^2} + \lambda^2 (1 - x^2) R = 0 , \tag{2-62}$$

and

$$\frac{d\lambda}{dx} = 0 . \tag{2-63}$$

In the BandShell style, a fragment of a DiffEQ function for this problem might be

```
begin
        case i of

        1:      DiffEQ := d2dx2(R,x) + sqr(l(x))*(1 - sqr(x)*R(x));   { = 0 }
        2:      DiffEQ := ddx(l,x)      { = 0 }

        end   { i cases }
end;
```

where R(x) represents $R(x)$ and l(x) represents $\lambda(x)$. Combined with suitable boundary conditions, this is a well-posed problem. Unfortunately, however, the central-difference approximation

to the derivative in the second equation leaves a row of zeros in the $[B_{i,k}]$ matrix. This problem can be avoided by rewriting the equation as follows

```
DiffEQ := I(x + h) - I(x);    { = 0 }
```

and, in general, this or a similar strategy must always be employed when solving for undetermined constants such a λ. (The complete text of a program to solve this eigenvalue problem is presented in a later section.)

Many times, problems with zero determinants and oscillations arise even when the first-order equation does not come from an undetermined constant, but rather from an actual first-order differential equation. These difficulties can be eliminated, and order h^2 accuracy for a central difference approximation can be retained, by writing the equation at the half-mesh point $(x + h/2)$, rather than on the point (x). By so doing, the finite-difference approximation looks like a forward difference

$$\frac{dc_k}{dx} = \frac{c_{k,j+1} - c_{k,j}}{h} \tag{2-64}$$

all non-derivative terms in the equation take on their values at the average between x and $x + h/2$, and the approximation retains its order-h^2 accuracy. Use of this approach for the eigenvalue problem discussed here yields:

```
DiffEQ := ddx(I,x + h/2);    { = 0 }
```

(This is exactly equivalent to the representation above involving difference of the values at adjacent mesh points, and both are accurate to order h^2.) The analogous relationships at $x - h/2$ (backward difference and averaging) may also be used, depending upon the circumstances.

2.8.9. Higher-Order Equations and Internal Flux-Matching Conditions

The BandShell procedure presented here is limited solely to systems of equations that result in a block-tridiagonal coefficient matrix (except for the boundary points). The major restrictions that this imposes on the method are that the derivatives in the equations must be no higher than second order, and that flux-matching conditions in the interior of the domain cannot be used.

The restriction to second-order equations is not a real limitation, since the problem can be avoided by creating, for a third-order equation, a set of two coupled second-order equations or a second-order equation and a first-order equation. The absence of flux-matching in the interior, however, is a true limitation, since it is likely to cause difficulties in problems containing two different regions. (This is often the case for electrochemical problems involving membrane separators or thin surface films.) An enhanced version of BAND that permits pentadiagonal entries can be written, but an alternative extension to the BAND algorithm that retains the block-tridiagonal form appears more attractive. This extension has been used [22,23], but it was decided not to include the feature in the present BandShell routine. Instead, it seems prudent to wait until more experience has been gained with this simpler version before adding the additional complexity to the software.

2.9. Solution of Partial Differential Equations: Time-Stepping

By combining the BAND algorithm with another method, partial differential equations can be solved. For example, a time-stepping technique can be applied to a limited class of partial differential equations, namely, those that are initial-value problems in one dimension and boundary-value problems in another. In this case, the partial differential equation problem can be reduced to the solution of a series of ordinary differential equations, which can be solved by BAND. The following example uses this technique to determine the unsteady-state concentration and potential profiles near a rotating disk electrode during linear sweep voltammetry.

If the problem is elliptic (*i. e.*, boundary-value problems in two dimensions), a combination of the BAND algorithm in one dimension and a collocation procedure in another is also possible. In this section, however, the discussion is restricted to time-stepping; a presentation of the use of collocation and BAND may be found in [24].

2.9.1. Mathematical Model of Linear Sweep Voltammetry

In dilute solutions, the flux of a species k, N_k, is generated by diffusion, convection, and migration:

$$N_k = -D_k \frac{\partial c_k}{\partial y} + v_y c_k - \frac{z_k D_k}{RT} c_k \frac{\partial \Phi}{\partial y}, \tag{2-65}$$

where, for a rotating disk electrode (in a solution of high Schmidt number),

$$v_y = -a\Omega\sqrt{\Omega/\nu}\, y^2, \tag{2-66}$$

and $a = 0.51023$.

Consequently, an unsteady-state material balance,

$$\frac{\partial c_k}{\partial t} = -\frac{\partial N_k}{\partial y}, \tag{2-67}$$

results in the following differential equation, a modified form of the convective-diffusion equation:

$$\frac{\partial c_k}{\partial t} = D_k \frac{\partial^2 c_k}{\partial y^2} - v_y \frac{\partial c_k}{\partial y} + \frac{z_k D_k}{RT}\left(c_k \frac{\partial^2 \Phi}{\partial y^2} + \frac{\partial c_k}{\partial y}\frac{\partial \Phi}{\partial y}\right) . \qquad (2\text{-}68)$$

The potential is determined implicitly by the electroneutrality condition on the species in solution:

$$\sum_k z_k c_k = 0 . \qquad (2\text{-}69)$$

Everywhere in the solution at initial time ($t = 0$), and far away from the disk ($y \rightarrow \infty$) at all times, the concentration retains its bulk value, $c_{k,o}$:

$$c_k(0,y) = c_k(t,\infty) = c_{k,o} . \qquad (2\text{-}70)$$

At the surface of the disk, the rate of reaction is determined by the Butler-Volmer kinetic expression, and the resulting flux is balanced by transport to the surface:

$$-\frac{nF}{s_k} N_k(0) = i_{o,ref}\left[\prod_k \left(\frac{c_k}{cko}\right)^{p_k} \exp(\,(1-\beta)\eta\,) - \prod_k \left(\frac{c_k}{cko}\right)^{q_k} \exp(\,-\beta\eta\,)\right] , \qquad (2\text{-}71)$$

where,

$$p_k = \begin{cases} s_k & \text{if } s_k > 0 \\ 0 & \text{if } s_k \leq 0 \end{cases} ,$$

$$q_k = \begin{cases} -s_k & \text{if } s_k < 0 \\ 0 & \text{if } s_k \geq 0 \end{cases} ,$$

$$\eta = \frac{nF}{RT}(V - \Phi(0) - U_o) ,$$

and the potential, V, is swept linearly at a rate of b volts per second:

$$V = U_o + bt .$$

For simplicity, the exchange-current density, $i_{o,ref}$, is based on reference concentrations equal to the bulk concentrations, $c_{k,o}$. (In general, this may not be the best choice, since bulk concentrations for some of the species may be zero.)

Finally, a zero of potential can be set at one point in the solution, chosen here to be y_{max}:

$$\Phi(y_{max}) = 0 \quad . \tag{2-72}$$

As with the vonKarman program earlier, the differential equations are defined on a semi-infinite domain, and y_{max} must be chosen to be large enough that the effect of y_{max} on the concentration distribution can be neglected. Its value is chosen as follows:

$$y_{max} = \left(\frac{3D_1}{a\nu}\right)^{1/3}\left(\frac{\nu}{\Omega}\right)^{1/2}\xi_{max} \tag{2-73}$$

where ξ_{max} is a maximum dimensionless distance from the disk surface. A value of ξ_{max} greater than about 1.6 is normally sufficient to ensure that the results are not affected [25].

Figure 2-11 shows the program required to calculate the profiles of $c_k(y)$ and $\Phi(y)$ at a number of discrete times during the sweep. The time derivatives are calculated by using the cInterp function with an argument of an array of values from a previous step, combined with another call with an argument of an array from the present step:

```
function dcdt( k : integer; x : RealNumber ) : RealNumber;
begin
        dcdt := ( cInterp(k,x,NewResult) - cInterp(k,x,OldResult) )/dt
end;
```

The definition of c:

```
function c( k : integer; x : RealNumber ) : RealNumber;
begin
        c := r*cInterp(k,x,NewResult) + (1 - r)*cInterp(k,x,OldResult)
end;
```

includes an implicit averaging between the steps, where r is the degree-of-implicitness. In the

program listing of figure 2-11, $r = 0.5$, which corresponds to equal averaging. It is known as the Crank-Nicolson symmetric form.

Figure 2-11. Text of Program LSV

```
[INHERIT('AidMod.pen','LSVIO.pen')]

program LSV( input, output );

const    Tolerance = 1e-6;
         F = 96487;   { coul/eq }
         RT = 2477.572,   { J/mol }
         r = 0.5;
         a = 0.51023;
         StepMax = 100;

var      m, jMax, j, k, ltMax, lt, CPUTime, Step : integer;
         D, z, s, co : Vector;
         Omega, nu, ioref, Uo, beta, n, b,
                  yMax, dt, t, tMax, UMax, dU, U, xMax : RealNumber;
         Guess, FinalResult, OldResult : ValueArray;


function Equation (
                  i, j : integer;

                  y, h : RealNumber;

                  var NewResult : ValueArray;

                  function clnterp ( k : integer;
                                     y : RealNumber;
                                     var Result : ValueArray ) : RealNumber;

                  function dnFdyn ( n : integer;
                                    function F ( y : RealNumber ) : RealNumber,
                                    y : RealNumber                ) : RealNumber

                                                                  ) : RealNumber;


         function c( k : integer; y : RealNumber ) : RealNumber;
         begin
                  c := r*clnterp(k,y,NewResult) + (1 - r)*clnterp(k,y,OldResult)
         end;


         function E( y : RealNumber ) : RealNumber;
         begin    E := clnterp(m+1,y,NewResult)      end;
```

```
function dcdy( k : integer; y : RealNumber ) : RealNumber;

        function ck( y : RealNumber ) : RealNumber;
        begin
                ck := c(k,y)
        end;

begin
        dcdy := dnFdyn(1,ck,y)
end;


function dcdt( k : integer; y : RealNumber ) : RealNumber;
begin
        dcdt := ( cInterp(k,y,NewResult) - cInterp(k,y,OldResult) )/dt
end;


function d2cdy2( k : integer; y : RealNumber ) : RealNumber;

        function ck( y : RealNumber ) : RealNumber;
        begin
                ck := c(k,y)
        end;

begin
        d2cdy2 := dnFdyn(2,ck,y)
end;


function dEdy( y : RealNumber ) : RealNumber;
begin
        dEdy := dnFdyn(1,E,y)
end;


function d2Edy2( y : RealNumber ) : RealNumber;
begin
        d2Edy2 := dnFdyn(2,E,y)
end;


function v( y : RealNumber ) : RealNumber;
begin
        v := -a*Omega*sqrt(Omega/nu)*sqr(y)
end,


function Flux( k : integer; y : RealNumber ) : RealNumber;

begin
        Flux := -D[k]*dcdy(k,y)

                + v(y)*c(k,y)

                - z[k]*( D[k]/RT )*F*c(k,y)*dEdy(y)

end,
```

```
function DiffEQ( k : integer, y : RealNumber ) : RealNumber;
begin

        DiffEQ := dcdt(k,y)

                - D[k]*d2cdy2(k,y) + v(y)*dcdy(k,y)

                - z[k]*( D[k]/RT )*F*( c(k,y)*d2Edy2(y)
                                        +  dcdy(k,y)*dEdy(y) )

                                                        { = 0 }

end;  { DiffEQ }


function Electroneutrality( y : RealNumber ) : RealNumber;

var     Sum : RealNumber;
        k : integer;

begin

        Sum := 0;

        for k := 1 to m do
                Sum := Sum + z[k]*c(k,y);

        Electroneutrality := Sum   { = 0 }

end;  { Electroneutrality }


function SurfaceBC( k : integer ) : RealNumber;

var     PiA, PiC, Eta : RealNumber;
        l : integer;


begin

        PiA := 1;
        PiC := 1;


        for l := 1 to m do
                if ( s[l] < 0 ) then
                        PiC := PiC*( c(l,0)/co[l] )**( -s[l] )
                else    PiA := PiA*( c(l,0)/co[l] )**( s[l] );


        Eta := ( n*F/RT )*( U - E(0) - Uo );


        if ( s[k] = 0 ) then

                SurfaceBC := Flux(k,0)   { = 0 }
```

```
                else
                        SurfaceBC := ( n/s[k] )*F*Flux(k,0)

                                + ioref*( PiA*exp( (1-beta)*Eta )
                                        - PiC*exp( -beta*Eta ) )

                                                                { = 0 }

        end;  { SurfaceBC }



begin  { body of Equation }

        if ( j = 1 ) then
                if ( i <= m ) then
                        Equation := SurfaceBC(i)
                else    Equation := Electroneutrality(0)

        else if ( j > 1 ) and ( j < jMax ) then
                if ( i <= m ) then
                        Equation := DiffEQ(i,y)
                else    Equation := Electroneutrality(y)

        else if ( j = jMax ) then
                if ( i <= m ) then
                        Equation := co[i] - c(i,yMax)  { = 0 }
                else    Equation := E(yMax)  { = 0 }

end;  { Equation }



begin  { body of LSV }

        ReadAndPrintParameters( m, jMax, ItMax, D, z, s, co,
                                Omega, nu, ioref, Uo, beta, n, b, dU, UMax, xMax );

        for j := 1 to jMax do
        begin
                for k := 1 to m do
                        Guess[k,j] := co[k];
                Guess[m+1,j] := Uo
        end;


        yMax := ( ( 3*D[1]/(a*nu) )**(1/3) )*sqrt(nu/Omega)*xMax;

        tMax := UMax/abs(b);
        dt := dU/abs(b);

        OldResult := Guess;
```

```
        It := 0;
        CPUTime := 0;
        t := 0;
        Step := 0;

        writeln; writeln;
        write('Step = ',Step:1,'      U = ',Uo:6:3,' V');
        PrintOut( OldResult, yMax, m, jMax, It, CPUTime );

        repeat
                t := t + dt;
                U := Uo + b*t;

                Step := Step + 1;


                BandAid( m+1, jMax, ItMax, It, CPUTime, yMax,
                        Tolerance, Equation, Guess, FinalResult );

                writeln; writeln;
                write('Step = ',Step:1,'   U = ',U:6:3,' V');
                PrintOut( FinalResult, yMax, m, jMax, It, CPUTime );

                OldResult := FinalResult;
                Guess := FinalResult


        until  ( t >= tmax - dt/2 ) or ( Step >= StepMax )

end.  { LSV }
```

2.10. Non-Uniform Mesh-Point Spacing: Coordinate Transformations

One of the principal limitations of the BandShell package is the requirement of uniform meshpoint spacing for all locations on the finite-difference grid. In some cases a single variable or a number of variables may change dramatically over a range that is small relative to the size of the domain, and, therefore, uniform spacing of the meshpoints may be inappropriate. The limitation can be overcome by creating a coordinate transformation, either to the independent variable or to one or another of the dependent variables, depending upon the situation. These coordinate transformations can be implemented quite easily by simple changes to the function definitions in the Equation subprogram, and the method for doing this is illustrated here.

The approach is always the same. First, the transformation equations for the variable are developed. Next, the chain rule is employed to determine new forms for the derivatives, and, finally, these relations are added to the code of the user program. (If necessary, the output routines can be adjusted to list the results in the original coordinate system.)

2.10.1. Coordinate Transformation for the Independent Variable

The most common occurrences of difficulties involve effects due to surfaces, which are normally on one end of the domain. For example, in the vonKarman program shown as the third example, the effect of the rotating disk dies out quickly as a function of distance from the surface. Consequently, a simple method to expand the coordinate system near the disk surface involves choosing a new independent variable that decays exponentially with distance from the disk. This new independent variable, x, can be defined as follows:

$$x = 1 - \exp(-\alpha z), \tag{2-74}$$

where z is the original coordinate, and x is the transformation. Thus, as x changes from 0 to 1, z covers the entire range from 0 to ∞. Asymptotic results for large ζ (presented in [18]) suggest that a value of 0.884 for α should be used so that the boundary conditions far from the disk will

be consistent with the original problem.

Application of the chain rule and substitution of the inverse transformation

$$z = -\frac{\ln(1-x)}{\alpha} \tag{2-75}$$

yields, for any function $F(z)$:

$$F(z) = F(x), \tag{2-76}$$

$$\frac{dF}{dz} = \alpha(1-x)\frac{dF}{dx}, \tag{2-77}$$

$$\frac{d^2F}{dz^2} = \alpha^2(1-x)\left[(1-x)\frac{d^2F}{dx^2} - \frac{dF}{dx}\right]. \tag{2-78}$$

Implementation of this transformation with the vonKarman program is straightforward. The meaning of the dependent variable is changed, and the definitions of the first-derivative operator and second-derivative operator are adjusted according to the transformation equations. Figure 2-12 shows the Equation function for the modified vonKarman program. Notice that all of the functions for the program are defined over the transformed domain x, since that is the domain of interest to BandShell, but that the equations retain their forms in the original coordinate system by virtue of the transformation functions. In this way, the equations remain documented in the source listing, and the coordinate transformation is explicitly indicated. The general relations for any transformation $x(z)$ are the following:

$$F(z) = F(x), \tag{2-79}$$

$$\frac{dF}{dz} = \frac{dF}{dx}\frac{dx}{dz}, \tag{2-80}$$

$$\frac{d^2F}{dz^2} = \frac{d^2F}{dx^2}\left[\frac{dx}{dz}\right]^2 + \frac{dF}{dx}\frac{d^2x}{dz^2}. \tag{2-81}$$

Figure 2-12. Text of Modified vonKarman Program

```
[INHERIT('AidMod.pen','KarmanIO.pen')]

program modKarman( input, output );

const    n = 4;
         alpha = 0.884;
         L = 1;
         Tolerance = 1e-6;

var      jMax, j, k, ItMax, It, CPUTime : integer;
         Guess, Result, Residual : ValueArray;


function Equation (
                  i, j : integer;

                  x, dx : RealNumber;

                  var NewResult : ValueArray;

                  function cInterp( k : integer;
                                    x : RealNumber;
                                    var Result : ValueArray ) : RealNumber;


                  function dnFdxn( n : integer;
                                   function F ( x : RealNumber ) : RealNumber,
                                   x : RealNumber                    ) : RealNumber

                                                                    ) : RealNumber;


         function F ( x : RealNumber ) : RealNumber;
         begin    F := cInterp(1,x,NewResult)       end;

         function G ( x : RealNumber ) : RealNumber;
         begin    G := cInterp(2,x,NewResult)       end;

         function H( x : RealNumber ) : RealNumber;
         begin    H := cInterp(3,x,NewResult)       end;

         function P( x : RealNumber ) : RealNumber;
         begin    P := cInterp(4,x,NewResult)       end;



         function ddz( function G( x : RealNumber ) : RealNumber;
                             x : RealNumber                 ) : RealNumber;
         begin
                  ddz := alpha*(1 - x)*dnFdxn(1,G,x)
         end;


         function d2dz2( function G( x : RealNumber ) : RealNumber;
                               x : RealNumber                 ) : RealNumber;
```

```
        begin
                d2dz2 := sqr(alpha)*(1-x)*( (1-x)*dnFdxn(2,G,x)
                                                - dnFdxn(1,G,x) )
        end;


        function EQ1( x : RealNumber ) : RealNumber;
        begin
                EQ1 := 2*F(x) + ddz(H,x)   { = 0 }
        end;


        function EQ2( x : RealNumber ) : RealNumber;
        begin
                EQ2 := sqr(F(x)) - sqr(G(x)) + H(x)*ddz(F,x) - d2dz2(F,x)   { = 0 }
        end;


        function EQ3( x : RealNumber ) : RealNumber;
        begin
                EQ3 := 2*F(x)*G(x) + H(x)*ddz(G,x) - d2dz2(G,x)   { = 0 }
        end;


        function EQ4( x : RealNumber ) : RealNumber;
        begin
                EQ4 := P(x) + sqr(H(x))/2 - 2*F(x)   { = 0 }
        end;


begin   { body of Equation }

        if ( j = 1 ) then
             case i of

             1:  Equation := H(0);          { = 0 }
             2:  Equation := F(0);          { = 0 }
             3:  Equation := G(0) - 1;      { = 0 }
             4:  Equation := EQ4(0)

             end   { i cases }

        else if ( j > 1 ) and ( j < jMax ) then
             case i of

             1:  Equation := EQ1(x - dx/2);
             2:  Equation := EQ2(x);
             3:  Equation := EQ3(x);
             4:  Equation := EQ4(x)

             end   { i cases }

        else if ( j = jMax ) then
             case i of

             1:  Equation := EQ1(L - dx/2);        { = 0 }
             2:  Equation := F(L);                 { = 0 }
             3:  Equation := G(L);                 { = 0 }
             4:  Equation := EQ4(L)
```

```
            end   { i cases }
end;  { Equation }


begin  { body of modKarman }
        ReadAndPrintParameters( jMax, ItMax );

        for j := 1 to jMax do
            for k := 1 to n do
            begin
                Guess[k,j] := 0
            end;

        BandAid( n, jMax, ItMax, It, CPUTime,
                            L, Tolerance, Equation, Guess, Result, Residual );

        PrintOut( Result, jMax, It, CPUTime, L )
end.  { modKarman }
```

If the transformation is presented directly in the form $z(x)$ (rather than $x(z)$ as above), the following relations can be used:

$$F(z) = F(x) , \tag{2-82}$$

$$\frac{dF}{dz} = \frac{dF}{dx}\left(\frac{dz}{dx}\right)^{-1} , \tag{2-83}$$

$$\frac{d^2F}{dz^2} = \frac{d^2F}{dx^2}\left(\frac{dz}{dx}\right)^{-2} - \frac{dF}{dx}\frac{d^2z}{dx^2}\left(\frac{dz}{dx}\right)^{-3} . \tag{2-84}$$

Notice that transformations $z(x)$ for which $\frac{dz}{dx} = 0$ at some point x in the domain are not permitted.

2.10.2. Coordinate Transformation for One or Another Dependent Variable

In other cases, one of the dependent variables may not vary linearly with the distance coordinate, and, therefore, it would be a very inaccurate representation of that variable to consider a linear variation between mesh points except for very large numbers of mesh points. In that event, it may be possible to consider a coordinate transformation of the dependent variable, if it can be determined what sort of variation is likely. Then, perhaps, the number of mesh points needed for accurate solution of the equations can be reduced.

This procedure can be applied to the concentration variable θ of the flow-through porous electrode example. Analysis of a simpler problem, namely that of complete mass-transfer control in the absence of diffusion and dispersion, indicates that the resulting concentration profile will simply be exponential decay over the distance of the reactor:

$$\theta = \exp(-y) . \tag{2-85}$$

The concentration can decay many orders of magnitude over the length of the reactor, and this is difficult to follow without a sufficiently large number of mesh points. Consequently, the appropriate variable for the problem is not concentration, θ, but rather the logarithm of concentration, $\ln(\theta)$, which will vary more linearly and less dramatically.[†] The potential, however, is not necessarily decaying in this manner, and therefore it need not be transformed.

The new variables for this problem are called u and v, and the original variables are related to them as follows:

$$\theta = \exp(u) , \tag{2-86}$$

$$\eta' = v . \tag{2-87}$$

[†] A transformation to the independent variable is another possibility in this case. A transformation to y, such as $x = 1 - \exp(-y)$, would result in a calling routine very similar to the modified vonKarman program of the previous section.

Application of the chain rule yields:

$$\frac{d\,\theta}{dy} = \exp(u)\,\frac{du}{dy}\,, \tag{2-88}$$

$$\frac{d\,\eta'}{dy} = \frac{dv}{dy}\,, \tag{2-89}$$

$$\frac{d^2\theta}{dy^2} = \exp(u)\left[\frac{d^2u}{dy^2} + \left(\frac{du}{dy}\right)^2\right], \tag{2-90}$$

and,

$$\frac{d^2\eta'}{dy^2} = \frac{d^2v}{dy^2}\,. \tag{2-91}$$

In general, the relations for a coordinate transformation $\theta(u)$ are the following:

$$\theta = \theta(u)\,, \tag{2-92}$$

$$\frac{d\,\theta}{dy} = \frac{d\,\theta}{du}\,\frac{du}{dy}\,, \tag{2-93}$$

$$\frac{d^2\theta}{dy^2} = \frac{d\,\theta}{du}\,\frac{d^2u}{dy^2} + \frac{d^2\theta}{du^2}\left(\frac{du}{dy}\right)^2\,. \tag{2-94}$$

The code for the Equation function of the modified FlowThru program is shown in figure 2-13. Since the difference approximation for the concentration variable θ is nonlinear, two possible representations for the flux term in the mass-balance equation are possible (as discussed in section 2.8.7). The first choice involves analytic differentiation of the expression for the divergence of the flux, resulting in the appearance of a second derivative, $\frac{d^2\theta}{dy^2}$, and a first derivative, $\frac{d\,\theta}{dy}$, both of which have a nonlinear form:

$$D'\,\frac{d^2\theta}{dy^2} - \frac{d\,\theta}{dy} = J_R\,. \tag{2-95}$$

The second choice uses a linear difference formula for the divergence, but keeps the nonlinear

form for the first derivative, $\frac{d\theta}{dy}$.

$$-ddy\left(-D' \frac{d\theta}{dy} + \theta \right) = J_R \ . \tag{2-96}$$

Use of the second choice guarantees that material balances on the reactant species will be satisfied for any number of meshpoints and, at the same time, preserves the exponential behavior incorporated in the coordinate transformation. This benefit of a mixed difference approximation for a material balance expression has been suggested in [17], where it was used specifically for this problem, and, therefore, it is the choice in program modFlow, shown in figure 2-13. Table 2-3 shows the effect of mesh size on the accuracy of the effluent concentration, and figure 2-14 compares the results of program modFlow to those obtained earlier from program FlowThru (section 2.8.4). The comparison shows that the results of both programs are accurate to order h^2.

The program listed in figure 2-13 contains essentially all of the features for the complete flow-through porous electrode program (presented in the appendix), which was used for the modeling in Part 1. Just as the program of figure 2-13, the complete program also includes the transformation of the dependent variable θ, but it extends the treatment shown here to include image points. In addition, it allows for the absence of the dispersion coefficient D' (which lowers the order of the equation) by writing the material-balance equation at half-mesh points, as suggested in section 2.8.7. Finally, the listing in the appendix also includes a (rather long) input/output module.

Figure 2-13. Text of Program modFlow

```
[INHERIT('AidMod.pen','FlowIO.pen')]

program modFlow( input, output );

const    n = 2;
         Tolerance = 1e-6;

var      P1, P2, P3, P4, DP, AlphaAR, AlphaCR,
                  AlphaAS, AlphaCS, AlphaL, IStar : RealNumber;
         jMax, j, ItMax, It, CPUTime : integer;
         Guess, Result, Residual : ValueArray;


function Equation (
                  i, j : integer;

                  y, h : RealNumber;

                  var NewResult : ValueArray;

                  function clnterp( k : integer;
                                    y : RealNumber;
                                    var Result : ValueArray ) : RealNumber;

                  function dnFdyn( n : integer;
                                   function F ( y : RealNumber ) : RealNumber;
                                   y : RealNumber                    ) : RealNumber

                                                                     ) : RealNumber;

         function u( y : RealNumber ) : RealNumber;
         begin    u := clnterp(1,y,NewResult)      end;

         function v( y : RealNumber ) : RealNumber;
         begin    v := clnterp(2,y,NewResult)      end;


         function ddy( function G( y : RealNumber ) : RealNumber;
                           y : RealNumber                        ) : RealNumber;
         begin
                  ddy := dnFdyn(1,G,y)
         end;


         function d2dy2( function G( y : RealNumber ) : RealNumber;
                             y : RealNumber                        ) : RealNumber;
         begin
                  d2dy2 := dnFdyn(2,G,y)
         end;


         function T( y : RealNumber ) : RealNumber;        {... Theta }
         begin    T := exp( u(y) )      end;


         function E( y : RealNumber ) : RealNumber;        {... Eta-Prime }
         begin    E := v(y)      end;
```

```
function dTdy( y : RealNumber ) : RealNumber;
begin
        dTdy := exp( u(y) )*ddy(u,y)
end;

function dEdy( y : RealNumber ) : RealNumber;
begin
        dEdy := ddy(v,y)
end;

function d2Tdy2( y : RealNumber ) : RealNumber;
begin
        d2Tdy2 := exp( u(y) )*( d2dy2(u,y) + sqr( ddy(u,y) ) )
end;

function d2Edy2( y : RealNumber ) : RealNumber;
begin
        d2Edy2 := d2dy2(v,y)
end;


function NR( y : RealNumber ) : RealNumber;    {... Reactant flux }
begin
        NR := -DP*dTdy(y) + T(y)
end;


function JR( y : RealNumber ) : RealNumber;    {... Main-reaction rate }

var     Top, Bottom : RealNumber;

begin
        Top := T(y) - P1*exp( ( (AlphaAR/AlphaCR) + 1 )*E(y) );
        Bottom := 1 + exp( E(y) );

        JR := Top/Bottom
end;


function JS( y : RealNumber ) : RealNumber;    {... Side-reaction rate }

var     exp1, exp2 : RealNumber;

begin
        exp1 := exp( -(AlphaCS/AlphaCR)*E(y) );
        exp2 := exp( ( (AlphaAS + AlphaCS)/AlphaCR )*E(y) );

        JS := P3*exp1*( 1 - P4*exp2 )
end;


function MassBalance( y : RealNumber ) : RealNumber;
begin
        MassBalance := -ddy(NR,y) - JR(y)    { = 0 }
end;


function ChargeBalance( y : RealNumber ) : RealNumber;
begin
```

```
                ChargeBalance := d2Edy2(y) - P2*( JR(y) + JS(y) )    { = 0 }
        end;


        function UpStrmBC( i : integer ) : RealNumber;
        begin
                case i of

                1:      UpStrmBC := T(0) - DP*dTdy(0) - 1;  { = 0 }
                2:      UpStrmBC := dEdy(0) + P2*IStar      { = 0 }

                end   { i cases }
        end;


        function DnStrmBC( i : integer ) : RealNumber;
        begin
                case i of

                1:      DnStrmBC := dTdy(AlphaL);   { = 0 }
                2:      DnStrmBC := dEdy(AlphaL)    { = 0 }

                end   { i cases }
        end;


begin   { body of Equation }

        if ( j = 1 ) then
                Equation := UpStrmBC(i)

        else if ( j > 1 ) and ( j < jMax ) then
                case i of
                1:      Equation := MassBalance(y);
                2:      Equation := ChargeBalance(y)
                end   { i cases }

        else if ( j = jMax ) then
                Equation := DnStrmBC(i)

end;   { Equation }


begin   { body of modFlow }

        ReadAndPrintParameters( P1, P2, P3, P4, DP, AlphaAR, AlphaCR,
                                AlphaAS, AlphaCS, AlphaL, IStar,
                                jMax, ItMax                          );

        for j := 1 to jMax do
        begin
                Guess[1,j] := 1;
                Guess[2,j] := 0
        end;

        BandAid( n, jMax, ItMax, It, CPUTime, AlphaL,
                        Tolerance, Equation, Guess, Result, Residual );

        for j := 1 to jMax do  Result[1,j] := exp( Result[1,j] );
```

```
        PrintOut( Result, jMax, It, CPUTime, AlphaL );
end  { modFlow }
```

Table 2-3. Effect of h^2 on $\theta(\alpha L)$ (program modFlow)

j_{max}	h	h^2	$\theta(\alpha L)$
26	0.34652	0.12006225	0.02032076
51	0.17326	0.03003289	0.04189855
101	0.08663	0.00750476	0.04798879
201	0.04332	0.00187619	0.04959208
401	0.02166	0.00046905	0.05000111

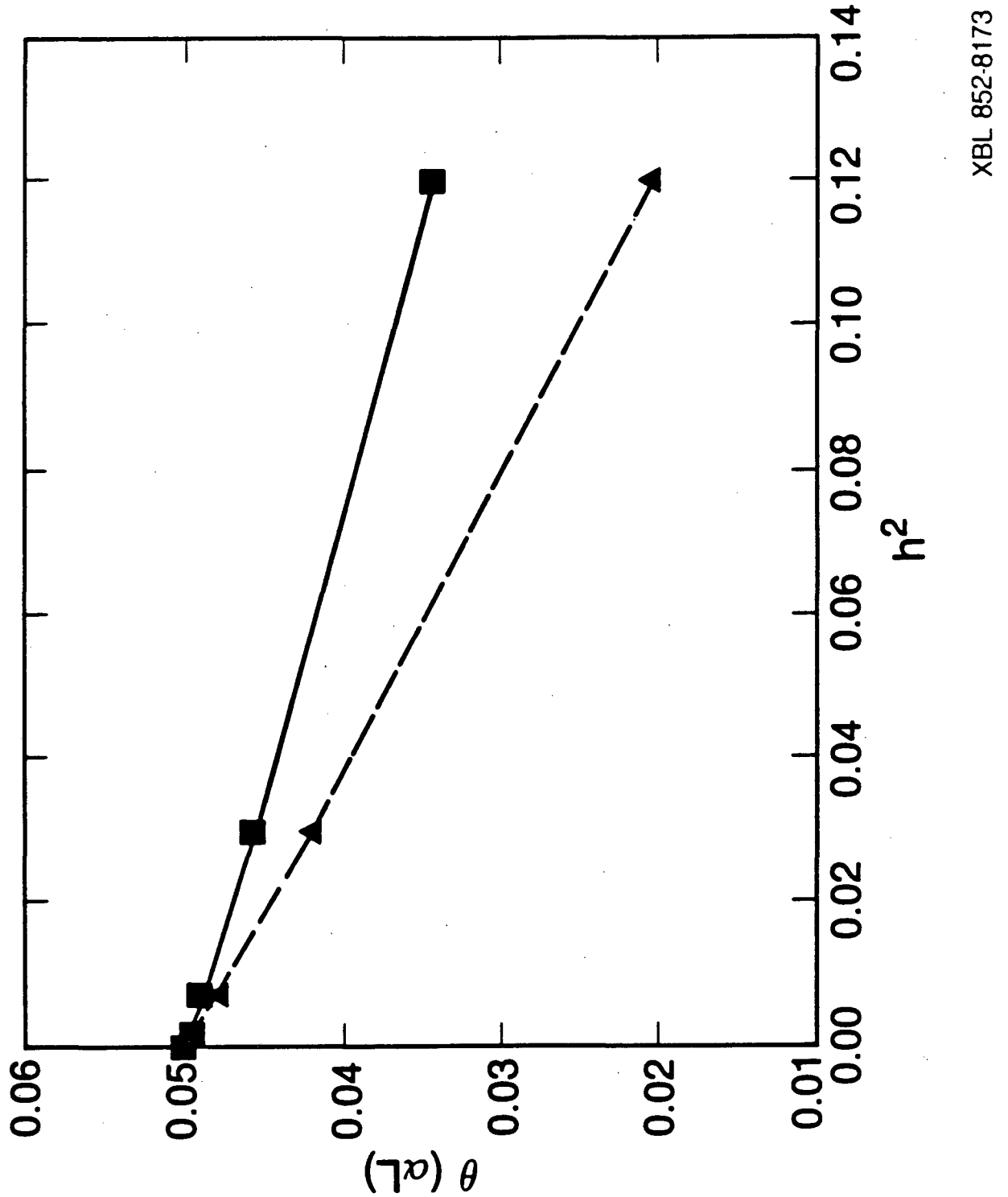


Figure 2-14. Effect of h^2 on the effluent concentration $\theta(\alpha L)$. The lower curve represents results from program modFlow, and the upper curve represents results from program FlowThru.

2.11. Use of the Complete BandShell Routine

Figure 2-15 shows the CatReac program of the first example, as it would be written for use with the complete BandShell routine.

2.11.1. Additional Parameters

Most of the parameters, such as n and xMax, are familiar from the BandAid case. The additional parameters that must be set here are ImageFirstPoint, ImageLastPoint, FactorIncrement, AbsoluteIncrement, ReduceTimeOption, and xMin.

The two (boolean) flags ImageFirstPoint and ImageLastPoint indicate if the boundaries of the domain contain image points, which are most often used to allow central-difference approximations to the derivatives to be used at all points in the domain. In the example, the need for image points is avoided by the use of three-point difference formulas at the boundaries (see the section on the BAND algorithm for details), and, therefore, both flags are set to "false." A program employing image points is presented later as a second example in this section.

As explained in the section on the BAND algorithm, linearization of the differential equations in the BandShell package is accomplished by numerical differentiation of the equations provided by the user. FactorIncrement and AbsoluteIncrement are employed by the routine to determine the increment ϵ used in that differentiation. When using the complete BandShell routine, these parameters must be set.

The ReduceTimeOption, which eliminates some unnecessary calculations in BandShell, is chosen in this example to be "true," which is the default used in BandAid. When "true," this option invokes a preprocessor in the BandShell routine that evaluates all of the equations defined in the Equation function under the conditions of the Guess. If a particular dependent variable does not appear at a given node point in a particular equation, the appropriate matrix entry (A, B, D, X, or Y) is flagged so that it is automatically set to zero for all subsequent calculations.

This is a very effective device, and it should be used whenever possible to increase the efficiency of the program. The user should be aware, however, of a possible pitfall when the Equation function contains conditional (**if**) statements. Since the flagging is only done once (with the initial guess), it is possible that something will be overlooked upon subsequent iteration. Therefore, it is advisable to set ReduceTimeOption to "false," whenever it is possible that the equations will be different from those present under the Guess. This is a relatively rare event, but it may occur if the kinetic rate law for a process is affected by the existence of a species (a precipitate, for example) that is not present under the conditions of the initial guess.

In the BandAid package, xMin is always set to zero. When using the complete BandShell routine, the value of xMin can be set to some other number. If, however, xMin is greater than xMax, an error message will be printed by BandShell on the standard output, and execution will be aborted.

The Equation function for BandShell is practically identical to the Equation function for BandAid. The only difference to be noted is the existence of an additional parameter, Approx, to the dnFdxn function. This parameter, which is of type DiffApprox, specifies the finite-difference approximation to be used in evaluating the derivative. Like ValueArray and RealNumber, the type DiffApprox is defined in a scope global to the routine. It is an enumerated type:

```
type    DiffApprox = ( CenDiff, BackDiff, ForDiff,
                                          Back3PtDiff, For3PtDiff );
```

In the current version of BandShell only five difference approximations are available with dnFdxn. The user is not obligated to use dnFdxn when describing equations, however, and other approximations to derivatives can be defined in a manner similar to the functions and composites if the need arises. In BandAid, since image points are not used, three-point difference formulas are used for boundary points, and central differences for points in the interior.

Figure 2-15. Listing of CatReac for Use With BandShell

```
[INHERIT('AidMod pen','CatIO pen')]

program fullCat( input, output );


const    n = 2;

         ImageFirstPoint = false,
         ImageLastPoint = false;
         FactorIncrement = 1e-6;
         AbsoluteIncrement = 1e-6;
         ReduceTimeOption = true;
         xMin = 0;

         Tolerance = 1e-6;


var      DA, DB, cAo, cBo, v, L, kf, kb : RealNumber;
         jMax, j, ItMax, It, Time : integer;
         Guess, Result, Deviation, Residual : ValueArray;


function Equation( i, j : integer;
                   x, h : RealNumber;
                   var NewResult : ValueArray;
                   function cInterp( k : integer;
                                     x : RealNumber;
                                     var Result : ValueArray ) : RealNumber;
                   function dnFdxn( n : integer;
                                    function F( x : RealNumber ) : RealNumber;
                                    x : RealNumber;
                                    Approx : DiffApprox ) : RealNumber
                                                          ) : RealNumber,

         function cA( x : RealNumber ) : RealNumber;
         begin    cA := cInterp(1,x,NewResult)     end;


         function cB( x : RealNumber ) : RealNumber;
         begin    cB := cInterp(2,x,NewResult)     end;


         function Difference : DiffApprox;
         begin
                  if ( j = 1 ) then
                           Difference := For3PtDiff

                  else if ( j = jMax ) then
                           Difference := Back3PtDiff

                  else     Difference := CenDiff
         end;
```

```
        function ddx( function G( x : RealNumber ) : RealNumber;
                          x : RealNumber                          ) : RealNumber;
        begin
                ddx := dnFdxn(1,G,x,Difference)
        end;

        function d2dx2( function G(x : RealNumber ) : RealNumber;
                            x : RealNumber                        ) : RealNumber;
        begin
                d2dx2 := dnFdxn(2,G,x,Difference)
        end;

        function RA( x : RealNumber ) : RealNumber;
        begin   RA := kf*cA(x)*cA(x) - kb*cB(x) end;

        function DiffEQ( i : integer; x : RealNumber ) : RealNumber;
        begin
                case i of

                1:  DiffEQ := DA*d2dx2(cA,x) - v*ddx(cA,x) - RA(x);  { = 0 }
                2:  DiffEQ := DB*d2dx2(cB,x) - v*ddx(cB,x) + RA(x)/2   { = 0 }

                end  { i cases }

        end;

        function BC1( i : integer ) : RealNumber;
        begin
                case i of

                1:  BC1 := -DA*ddx(cA,0) + v*( cA(0) - cAo );  { = 0 }
                2:  BC1 := -DB*ddx(cB,0) + v*( cB(0) - cBo )   { = 0 }

                end  { i cases }
        end;

        function BC2( i : integer ) : RealNumber;
        begin
                case i of

                1:  BC2 := ddx(cA,L);    { = 0 }
                2:  BC2 := ddx(cB,L)     { = 0 }

                end  { i cases }
        end;


begin  { body of Equation }

        if ( j = 1 ) then
                Equation := BC1(i)

        else if ( j > 1 ) and ( j < jMax ) then
                Equation := DiffEQ(i,x)
```

```
        else if ( j = jMax ) then
                Equation := BC2(i)

end;  { Equation }



function Converged( function x( Node : integer ) : RealNumber;
                    h : RealNumber;
                    var NewResult, Deviation : ValueArray;
                    var Residual : ValueArray;
                    function cInterp( k : integer;
                                      x : RealNumber;
                                      var Result : ValueArray ) : RealNumber;
                    function dnFdxn( n : integer;
                                     function F( x : RealNumber ) : RealNumber;
                                     x : RealNumber;
                                     Approx : DiffApprox ) : RealNumber
                                                                ) : boolean;

var     k, j : integer;
        absDeviation, absValue : RealNumber;

begin

        Converged := true;

        for j := 1 to jMax do
                for k := 1 to n do

                begin
                        absDeviation := abs(Deviation[k,j]);
                        absValue := abs(NewResult[k,j]);

                        if ( absValue <> 0 ) then
                            if ( absDeviation > Tolerance*absValue ) then
                                    Converged := false

                end

end;  { Converged }


procedure NonBandCalcs( LastIteration : boolean;
                        Iteration, CPUTime : integer;
                        function x( j : integer ) : RealNumber;
                        h : RealNumber;
                        var NewResult, Deviation : ValueArray;
                        var Residual : ValueArray;
                        function cInterp( k : integer;
                                          x : RealNumber;
                                          var Result : ValueArray ) : RealNumber;
                        function dnFdxn( n : integer;
                                         function F( x : RealNumber ) : RealNumber;
                                           x : RealNumber;
                                           Approx : DiffApprox ) : RealNumber );
```

```
begin

        if LastIteration then
        begin
                It   := Iteration;
                Time := CPUTime
        end

end.  { NonBandCalcs }
```

```
begin  { body of fullCat }

        ReadAndPrintParameters( DA, DB, cAo, cBo, v, L, kf, kb, jMax, ItMax );

        for j := 1 to jMax do
        begin
                Guess[1,j] := cAo;
                Guess[2,j] := cBo
        end;

        BandShell( n, jMax, ItMax,
                   xMin, L, FactorIncrement, AbsoluteIncrement,
                   ImageFirstPoint, ImageLastPoint, ReduceTimeOption,
                   Guess, Result, Deviation, Residual,
                   Equation, Converged, NonBandCalcs );


        PrintOut( Result, jMax, It, Time, L )

end.  { fullCat }
```

In addition to the Equation function, use of the complete BandShell routine requires an additional function subprogram, Converged. The structure of this routine is very similar to Equation, but whereas Equation returns a RealNumber value representing a residual, Converged returns a boolean value ("true" or "false") indicating whether the system of equations is satisfied to within some specified convergence criteria. Since, in general, the criteria for convergence may

involve relatively complicated functional descriptions, the same sort of format as for the Equation function is provided. In BandAid, a simple form for the convergence is employed for all cases.

The final additional parameter needed for the complete BandShell routine is the procedure subprogram NonBandCalcs. The most common use of NonBandCalcs is for calculations of derived quantities, such as current, which, although not part of the BAND calculations, can be facilitated by use of either cInterp or dnFdxn. NonBandCalcs is used, because cInterp and dnFdxn are local to the Equation function and, therefore, cannot be accessed by other parts of the user program.

In addition, NonBandCalcs acts as a kind of safety valve for all of the things that have been left out or overlooked in the general structure of BandShell. For more sophisticated applications, where, for example, manipulations of the intermediate results of the BandShell calculations need to be made, the NonBandCalcs routine provides a mechanism for the user program to get inside of BandShell at each iteration for these special manipulations. Since extensive use of NonBandCalcs for this purpose is quite dangerous, the BandAid routine has no such mechanism. If necessary, the user program can examine the results after each iteration without using NonBandCalcs by "manual" iteration, *i. e.*, by setting the ItMax parameter to 1, and by placing the BandAid or BandShell call within a loop. The major disadvantage of this approach is that it requires that the user program check for convergence. In NonBandCalcs, the boolean variable LastIteration can be used, as it is in the CatReac example, to indicate when a converged solution has been found. The integer variable Iteration indicates the current iteration, so that, for example, the intermediate results every fifth iteration can be printed. The integer CPUTime indicates an approximate execution time (in milliseconds) as calculated from the Clock function available in VAX-11 Pascal.

Use of the BandShell routine is illustrated in the following example, calculation of eigenvalues and eigenfunctions for the asymmetric Graetz problem. In addition, the appendix contains a listing of the BandAid code, which involves a simple call to BandShell, very similar to the CatReac program shown above.

2.11.2. Eigenvalues and Eigenfunctions for the Asymmetric Graetz Problem

A complete program for calculating the eigenvalues and eigenfunctions for the asymmetric Graetz problem introduced earlier is presented in figure 2-16. This is the first example to use an image point (at j_{max}), and the only example wherein the result depends strongly on the choice of initial guess. The problem statement is the following [21]:

$$\frac{d^2R}{dx^2} + \lambda_m^2(1 - x^2)R = 0 , \tag{2-97}$$

$$\frac{d\lambda_m}{dx} = 0 , \tag{2-98}$$

where,

$$\text{at } x = -1 \qquad R = 0 , \tag{2-99}$$

$$\text{at } x = 1 \qquad R = 0 \text{ and } \frac{dR}{dx} = 1 . \tag{2-100}$$

Notice that there are three boundary conditions on R, but that there are no boundary conditions on λ_m. The program uses an image point at $x = x_{max}$ and applies the boundary conditions as well as the governing differential equations at that position. In this way, the value of λ_m is determined implicitly by the program through the interaction of the equations and boundary conditions without any algebraic manipulations independent of the solution of the rest of the problem.

Notice that care must be taken when using image points at boundaries in order that problems of oscillations and zero determinants do not arise. Notice also that the equations at $j = j_{max}$ and at $j = j_{max} - 1$ are all applied at $x = x_{max}$, since, strictly speaking, none of the equations is valid outside of the domain. The user must be careful to state this explicitly because, in fact, $x(j_{max})$ is not x_{max} in this case, but, rather, $x_{max} + h$.

One final complication arising in this problem is the existence of a strong dependence of the resulting eigenvalues and eigenfunctions on the initial guess. In general, the boundary-value

problems solved with the BAND method involve physical systems for which only a unique solution is possible. In this case, however, because it is an eigenvalue problem, there is an infinite number of possible solutions, one corresponding to each of the eigenvalues λ_m. Since this eigenvalue problem is very similar to the eigenvalue problem whose solution is

$$R = \frac{\sin((1-x)\lambda_m)}{\lambda_m}, \qquad \lambda_m = \frac{m\pi}{2}, \tag{2-101}$$

the eigenvalues and eigenfunctions for that problem are guessed initially here. A more comprehensive discussion of the use of BAND to solve this sort of eigenvalue problem can be found in [26].

Figure 2-16. Text of Program Graetz

```
[INHERIT('AidMod.pen','GraetzIO.pen')]

program Graetz( input, output );

const    n = 2;

         ImageFirstPoint = false;
         ImageLastPoint = true;
         FactorIncrement = 1e-6;
         AbsoluteIncrement = 1e-6;
         ReduceTimeOption = true;
         xMin = -1;
         xMax = 1;

         Tolerance = 1e-10;
         pi = 3.141592654;

var      Result, Guess, Deviation, Residual : ValueArray;
         j, jMax, It, ItMax, Time : integer;
         h, x : RealNumber;
         m : integer;

function Equation( i, j : integer;
                   x, h : RealNumber;
                   var NewResult : ValueArray;
                   function cInterp( k : integer;
                                     x : RealNumber;
                                     var Result : ValueArray ) : RealNumber;
```

```
                function dnFdxn( n : integer;
                                 function F( x : RealNumber ) : RealNumber,
                                 x : RealNumber;
                                 Approx : DiffApprox ) : RealNumber
                                                       ) : RealNumber;

        function R( x : RealNumber ) : RealNumber,
        begin   R := cInterp(1,x,NewResult)      end;

        function I( x : RealNumber ) : RealNumber,
        begin   I := cInterp(2,x,NewResult)      end;

        function ddx( function G( x : RealNumber ) : RealNumber;
                        x : RealNumber             ) : RealNumber;
        begin
                ddx := dnFdxn(1,G,x,CenDiff)
        end;

        function d2dx2( function G( x : RealNumber ) : RealNumber;
                          x : RealNumber             ) : RealNumber;
        begin
                d2dx2 := dnFdxn(2,G,x,CenDiff)
        end;

        function EQ1( x : RealNumber ) : RealNumber;
        begin
                EQ1 := d2dx2(R,x) + sqr( I(x) )*(1 - sqr(x))*R(x)    { = 0 }
        end;

        function EQ2( x : RealNumber ) : RealNumber;
        begin
                EQ2 := I(x) - I(x + h)    { = 0 }
        end;


begin  { body of Equation }

        if ( j = 1 ) then
            case i of
            1:          Equation := R(-1);       { = 0 }
            2:          Equation := EQ2(-1)      { = 0 }
            end  { i cases }

        else if ( j > 1 ) and ( j < jMax - 1 ) then
            case i of
            1:          Equation := EQ1(x);      { = 0 }
            2:          Equation := EQ2(x)       { = 0 }
            end  { i cases }

        else if ( j = jMax - 1 ) then
            case i of
            1:          Equation := EQ1(1);      { = 0 }
```

```
            2:          Equation := R(1)          { = 0 }
            end  { i cases }

        else if ( j = jMax ) then
            case i of
            1:          Equation := ddx(R,1) - 1;        { = 0 }
            2           Equation := EQ2(1)                 { = 0 }
            end  { i cases }


end;  { Equation }



function Converged( function x( Node : integer ) : RealNumber;
                    h : RealNumber;
                    var NewResult, Deviation : ValueArray;
                    var Residual : ValueArray;
                    function cInterp( k : integer;
                                      x : RealNumber;
                                      var Result : ValueArray ) : RealNumber;
                    function dnFdxn( n : integer;
                                     function F( x : RealNumber ) : RealNumber;
                                     x : RealNumber;
                                     Approx : DiffApprox ) : RealNumber
                                                                ) : boolean;

var     k, j : integer;
        absDeviation, absValue : RealNumber;

begin

        Converged := true;

        for j := 1 to jMax do
                for k := 1 to n do

                begin
                        absDeviation := abs(Deviation[k,j]);
                        absValue := abs(NewResult[k,j]);

                        if ( absValue <> 0 ) then
                            if ( absDeviation > Tolerance*absValue ) then
                                    Converged := false

                end

end;  { Converged }


procedure NonBandCalcs( LastIteration : boolean;
                        Iteration, CPUTime : integer;
                        function x( j : integer ) : RealNumber;
                        h : RealNumber;
                        var NewResult, Deviation : ValueArray;
                        var Residual : ValueArray;
```

```
                        function cInterp( k : integer;
                                          x : RealNumber;
                                          var Result : ValueArray ) : RealNumber;
                        function dnFdxn( n : integer;
                                         function F( x : RealNumber ) : RealNumber;
                                             x : RealNumber;
                                             Approx : DiffApprox ) : RealNumber );

begin

        if LastIteration then
        begin
                It := Iteration;
                Time := CPUTime
        end

end;  { NonBandCalcs }


begin  { body of Graetz }

        ReadAndPrintParameters( m, jMax, ItMax );

        It := 0;

        h := 2/(jMax - 2);

        for j := 1 to jMax do
        begin
                x := -1 + (j - 1)*h;

                Guess[1,j] := -sin( (1 - x)*(pi*m/2) )/( pi*m/2 );

                Guess[2,j] := pi*m/2
        end;


        BandShell( n, jMax, ItMax,
                   xMin, xMax, FactorIncrement, AbsoluteIncrement,
                   ImageFirstPoint, ImageLastPoint, ReduceTimeOption,
                   Guess, Result, Deviation, Residual,
                   Equation, Converged, NonBandCalcs );


        PrintOut( Result, jMax, It, Time, xMin, xMax )


end.  { Graetz }
```

2.12. Conclusions and Perspectives on Future Work

The programs in this document are models of a functional programming style that has been developed for solving one-dimensional boundary-value problems by finite-difference techniques. This style produces efficient, reliable, and elegant computer programs, and, combined with the supporting BandAid procedure, it facilitates greatly the development of sophisticated mathematical models of physical processes. In the next few years, as the technology of very large scale integration (VLSI) advances, supercomputers containing many thousands of processors operating in parallel will become available. Concurrent aspects of Newman's algorithm can be used to take advantage of this increased computing power, and, with this extension, the non-sequential format advocated here will be particularly useful. In addition, although this manuscript concentrates on finite-difference procedures, the overall approach is general, and it is likely that the methods shown here could be used effectively with other numerical techniques, such as collocation.

Acknowledgements

Special thanks are due to Mr. Bill Hogan for many valuable discussions regarding this work and to Mr. John Cain and Mr. Paul Shain for their willingness to work with earlier (incomplete) versions of BandAid.

List of Symbols

a	Proportionality constant
a	Proportionality constant in the expression for the normal velocity to a rotating disk, 0.51023
A	Reactant species in catalytic reactor problem
$A_{i,k,j}$	Matrix coefficient in the BAND algorithm, $\frac{\partial F_i}{\partial c_{k,j-1}}$
B	Product species in the catalytic reactor problem
b	Sweep rate for linear sweep voltammetry problem, V/s
$B_{i,k,j}$	Matrix coefficient in the BAND algorithm, $\frac{\partial F_i}{\partial c_{k,j}}$
c	Concentration, mol/cm^3
c_A	Concentration of species A, mol/cm^3
$c_{A,o}$	Feed concentration of A, mol/cm^3
c_B	Concentration of species B, mol/cm^3
$c_{B,o}$	Feed concentration of B, mol/cm^3
c_k	Concentration of k^{th} species, mol/cm^3
$c_{k,j}$	Value of k^{th} variable at meshpoint j
$c^*_{k,j}$	Trial value of k^{th} variable at meshpoint j
$c_{k,o}$	Bulk concentration of species k, mol/cm^3
D	Diffusion coefficient, cm^2/s
D_A	Diffusion coefficient of species A, cm^2/s
D_B	Diffusion coefficient of species B, cm^2/s
$\Delta c_{k,j}$	Deviation of the value of $c_{k,j}$ from trial conditions $c^*_{k,j}$

$D_{i,k,j}$	Matrix coefficient in BAND algorithm, $\frac{\partial F_i}{\partial c_{k,j+1}}$
D_k	Diffusion coefficient of species k, cm^2/s
D_o	Diffusion coefficient at infinite dilution, cm^2/s
D'	Dimensionless dispersion coefficient in flow-through porous electrode model
$E_{k,i,j}$	Entry in decomposition matrix used in the BAND algorithm
F	An arbitrary continuous function of x
F	Faraday's constant, 96487 coul/eq
F	Radial velocity in the von Kármán transformation for flow to a rotating disk
F_i	Residual of the i^{th} equation
$\bar{F}_i$	Linearized form of residual F_i
F_i^o	Value of residual at trial conditions $c_{k,j}^o$
G	Angular velocity in the von Kármán transformation for flow to a rotating disk
G_i	Entry in the vector of non-homogeneous terms for the matrix equations of the BAND algorithm, $-F_i^o$
h	Mesh size
H	Axial velocity in the von Kármán transformation for flow to a rotating disk
i	Index for equation numbers
$i_{o,ref}$	Reference exchange-current density, A/cm^2
I^*	Dimensionless current density to porous electrode
j	Index for meshpoints
j_{max}	Number of meshpoints
J_R	Dimensionless rate term for the main reaction in the porous electrode

J_S	Dimensionless rate term for the side reaction in the porous electrode
k	Index for dependent variables
k_b	Rate constant for reverse reaction in catalytic reactor, s^{-1}
k_f	Rate constant for forward reaction in catalytic reactor, $cm^3/mol-s$
L	Length of catalytic reactor, cm
L	Porous electrode reactor length, cm
m	Index for eigenvalues
N	Molar flux, mol/cm^2-s
N	Iteration number
N_R	Dimensionless reactant-ion flux
n	Number of electrons transferred in an electrode reaction
n	Number of equations and unknowns
N_k	Molar flux of species k, mol/cm^2-s
P	Pressure variable in the von Kármán transformation for flow to a rotating disk
P	Pressure, $dyne/cm^2$
P_1	Parameter characterizing the backward term of the main reaction in the flow-through porous electrode
P_2	Parameter characterizing the relative importance of ohmic resistance to mass-transfer resistance in the flow-through porous electrode
P_3	Parameter characterizing the rate of side reaction in the flow-through porous electrode
P_4	Parameter characterizing the backward term in the side reaction in the flow-through porous electrode
p_k	Exponent in the composition dependence of the anodic term in the Butler-Volmer expression

q_k	Exponent in the composition dependence of the cathodic term in the Butler-Volmer expression
r	Degree of implicitness for time-stepping problems
R	Universal gas constant, 8.314 J/mol-K
R	Graetz function
r	Radial distance, cm
R_A	Rate of disappearance of species A, mol/cm^3-s
s_k	Stoichiometric coefficient for species k
T	Temperature, K
t	Time, s
u	Transformed dependent variable
U_o	Reference potential, at bulk concentrations of all species, V
V	Electrode potential, V
v	Transformed dependent variable
v	Velocity of fluid through reactor, cm/s
v_r	Radial velocity, cm/s
v_θ	Angular velocity at rotating disk, cm/s
v_y	Normal velocity to rotating disk, cm/s
v_z	Axial velocity, cm/s
x	Distance through reactor, cm
x	Distance variable in the BAND algorithm
x_j	Distance corresponding to meshpoint j
x_{max}	Maximum value of independent variable $= x(j_{max})$
$X_{i,k,j}$	Matrix coefficient in BAND algorithm, $\frac{\partial F_i}{\partial c_{k,j+2}}$
y	Dimensionless distance through porous electrode

y	Normal distance from surface of rotating disk, cm
y_{max}	Maximum axial distance from rotating disk, cm
$Y_{i,k,j}$	Matrix coefficient in BAND algorithm, $\frac{\partial F_i}{\partial c_{k,j-2}}$
z	Axial distance
z	Distance variable, cm
z_k	Valence of k^{th} species
α_{aR}	Anodic transfer coefficient for the main reaction in the porous electrode
α_{aS}	Anodic transfer coefficient for the side reaction in the porous electrode
α_{cR}	Cathodic transfer coefficient for the main reaction in the porous electrode
α_{cS}	Cathodic transfer coefficient for the side reaction in the porous electrode
α	Reciprocal of penetration depth into the porous electrode, cm^{-1}
αL	Dimensionless length of flow-through porous electrode
β	Symmetry coefficient in the Butler-Volmer expression
ϵ	Increment used for numerical differentiation in BandCore
ς	Dimensionless axial distance
ς_{max}	Maximum axial distance
η	Potential driving force
η'	Dimensionless potential driving force in flow-through porous electrode model
θ	Dimensionless reactant ion concentration
λ	Eigenvalue in the asymmetric Graetz problem

λ_m	m^{th} eigenvalue in the asymmetric Graetz problem
μ	Viscosity, g/cm-s
ν	Kinematic viscosity, cm^2/s
$\xi_{k,j}$	Entry in decomposition vector used in the BAND algorithm
ξ_{max}	Maximum dimensionless distance from surface of rotating disk
Φ	Potential, V
Ω	Rotation speed, rad/s

References

1. John Newman, "Numerical Solution of Coupled, Ordinary Differential Equations (UCRL-17739)," Lawrence Radiation Laboratory, University of California, Berkeley (August, 1967).

2. John Newman, "Numerical Solution of Coupled, Ordinary Differential Equations," *Industrial and Engineering Chemistry Fundamentals*, **7** (1968), 514-517.

3. John Newman, *Electrochemical Systems*, (Englewood Cliffs, NJ: Prentice Hall, Inc., 1973), Appendix C, 414-425.

4. Peter Henderson, *Functional Programming: Application and Implementation*, (London: Prentice-Hall International, Inc., 1980).

5. Doug Cooper and Michael Clancy, *Oh! Pascal*, (New York: W. W. Norton & Company, Inc., 1982).

6. Doug Cooper, *Standard Pascal User Reference Manual*, (New York: W. W. Norton & Company, Inc., 1983).

7. Niklaus Wirth, "Program Development by Stepwise Refinement," *Communications of the Association for Computing Machinery*, **14** (1971), 221-227.

8. Niklaus Wirth, *Systematic Programming: An Introduction*, (Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1973).

9. Niklaus Wirth, *Algorithms + Data Structures = Programs*, (Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1976).

10. Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, (New York: Springer-

Verlag, 1974).

11. Kathleen Jensen, *Why Pascal?*, (Digital Equipment Corporation, 1979).

12. Brian W. Kernighan, *Why Pascal Is Not My Favorite Programming Language*, (Computing Science Technical Report No. 100, Bell Telephone Laboratories, Inc., July 18, 1981).

13. Brian W. Kernighan and P. J. Plauger, *The Elements of Programming Style*, (New York: McGraw-Hill, Inc., 1978).

14. Henry F. Ledgard, Paul A. Nagin, and John F. Hueras, *Pascal With Style -- Programming Proverbs*, (Rochelle Park, NJ: Hayden Book Company, 1979).

15. J. F. Wehner and R. H. Wilhelm, "Boundary Conditions of A Flow Reactor," *Chemical Engineering Science*, **6** (1956), 89.

16. P. V. Danckwerts, "Continuous Flow Systems. Distribution of Residence Times," *Chemical Engineering Science*, **12** (1978), 1.

17. James A. Trainham and John Newman, "A Flow-Through Porous Electrode Model: Application to Metal-Ion Removal from Dilute Streams," *Journal of the Electrochemical Society*, **124** (1977), 1528-1540.

18. Ralph White, Charles M. Mohr, Peter Fedkiw, and John Newman, *The Fluid Motion Generated by a Rotating Disk: A Comparison of Solution Techniques*, LBL-3910, Lawrence Berkeley Laboratory (November, 1975).

19. Th. von Kármán, "Uber laminare und turbulente Reibung," *Zeitschrift für angewandte Mathematik und Mechanik*, **1** (1921), 233-252.

20. Douglas N. Bennion, "Modeling and Reactor Simulation," in *AIChE Symposium Series*, **79**.

Number 229, *Tutorial Lectures in Electrochemical Engineering and Technology -- II*, Richard Alkire and Der-Tau Chin, editors, (New York: American Institute of Chemical Engineers, 1983), 25-36.

21. Victoria Edwards and John Newman, "The Asymmetric Graetz Problem in Channel Flow," submitted to the *International Journal of Heat and Mass Transfer*.

22. Ralph White, *Simultaneous Reactions on a Rotating-Disk Electrode*, dissertation, University of California, Berkeley (1977), LBL-6094, Appendix D.

23. Philip P. Russell, *Corrosion of Iron: The Active-Passive Transition and Sustained Electrochemical Oscillations*, dissertation, University of California, Berkeley (1984).

24. Peter S. Fedkiw, "A Collocation-Finite Difference Procedure for an Elliptic Partial Differential Equation," *Computers and Chemical Engineering*, **6** (1982), 327-330.

25. John Newman, *Electrochemical Systems*, (Englewood Cliffs, NJ: Prentice Hall, Inc., 1973), p. 306.

26. John Newman, "The Fundamental Principles of Current Distribution and Mass Transport in Electrochemical Cells," *Electroanalytical Chemistry*, **6**, Allen J. Bard, ed., 196-221 (New York: Marcel Dekker, 1973).

Part 3. Use of Duhamel's Superposition Integral for Numerical Simulations of Transient Current Responses in Electrochemical Systems

3.1. Summary

A fast, simple algorithm is presented for simulating the transient current response of complex reaction systems to arbitrary applied-potential waveforms. Although the basic algorithm is valid for many waveforms and for a number of electrode geometries, the programs developed here emphasize the computation of linear sweep voltammograms (ramps) and of cyclic voltammograms (triangular waves) at planar electrodes and at rotating disk electrodes. Development of a general-purpose software package is discussed, and simulated voltammograms for two electrochemical systems are compared to experiment. These sample simulations indicate that the method is an efficient calculating procedure for a wide variety of electrochemical systems. Listings of the computer programs used for the sample calculations are contained in the appendix.

3.2. Introduction

Application of a potential waveform, such as a step function or a square wave, to an electrochemical system and observation of the resulting current response can be an effective means for examining the nature of reactions occurring at an electrode surface. Close agreement between the actual current response of an electrochemical system and the response predicted from a particular mechanism provides strong evidence for the validity of a reaction pathway.

Although the observation of surface reactions is the primary purpose of current-response measurements, the electrodes most often used (planar electrodes and rotating disk electrodes) provide only a small contact area with the electrolyte solution, and, as a result, concentration gradients can develop. Because of these concentration gradients, the rate of mass transfer of electroactive species to the electrode surface must be taken into account when examining

experimental results for mechanistic information, and theoretical investigations of current-response experiments are necessarily concerned with the mathematical analysis of mass-transfer processes near electrochemical surfaces. In the mathematical analysis, the information of interest, the electrochemical reaction mechanism, is normally incorporated as a boundary condition in the appropriate mass-transfer problem.

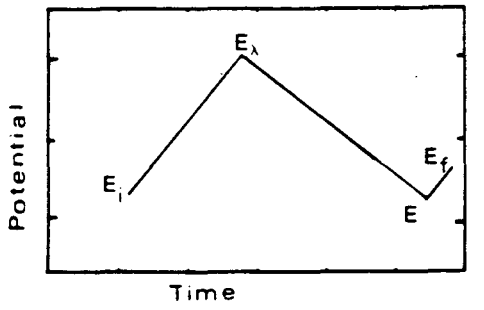
One of the most common types of current-response techniques is cyclic voltammetry,[†] in which the applied-potential waveform is a triangular wave. Cyclic voltammetry is a popular method, since triangular waveforms are easy to generate and to control and the resulting potential is a simple function of time. An example of a typical triangular wave is shown in figure 3-1a, and a possible current response for an electrochemical system is shown in figure 3-1b as a function of time. Often, the two curves are combined, as illustrated in figure 3-1c, to form a plot known as a cyclic voltammogram. In the cyclic voltammogram, each complete cycle of applied potential forms a single loop, and since time is not shown on the coordinate axes, the sweep rate (slope of the potential ramp, V/s) is generally indicated to complete the description. Characteristic peaks and waves in the current response mark the electrochemical reactions occurring on the electrode surface, and these peaks and waves can be interpreted, qualitatively or quantitatively, to determine possible reaction mechanisms.

A rich literature is available describing the basic foundations of cyclic-voltammetry theory, and complete analytic solutions to mass-transfer problems have been developed for a number of reaction mechanisms, usually (but not always) restricted to simple (1:1) reaction stoichiometries, reversible electrode reactions, or Tafel kinetics. A short review of the important contributions of Nicholson and Shain [2] and of Shuman [3] can be found in [4], and more complete reviews and discussions of the technique can be found in [1,5].

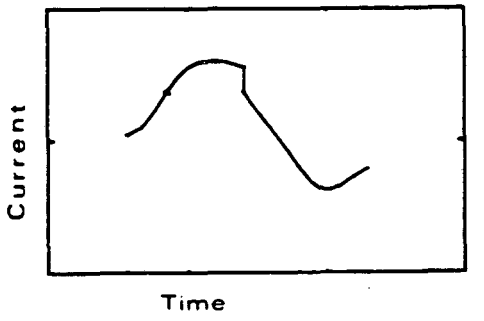
The results obtained by these workers and others are very useful, since analytic solutions often indicate specific diagnostic criteria that can be employed easily to check a simple reaction

---

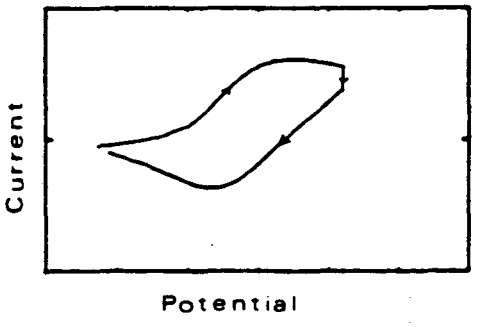
[†] Also known as triangular potential-sweep chronoamperometry [1].



a) Applied Potential



b) Current



c) Voltammogram

XBL 847-3081

Figure 3-1. Typical waveforms in cyclic voltammetry. (3-1a) Potential as a function of time. (3-1b) Current response as a function of time. (3-1c) Current response as a function of potential.

mechanism. In addition, an understanding of the characteristic limiting behavior of simple systems must be obtained before complicated systems can be investigated. Nevertheless, the mathematical complexity corresponding to the analysis of complicated reaction mechanisms often requires that numerical techniques be employed to evaluate the voltammograms. These numerical solutions are not as general as analytic results, since they do not reveal universal diagnostic criteria, but they are valuable when the use of analytic results would involve too much approximation or oversimplification of the mechanism.

Among the methods available for numerical analysis, the use of superposition integrals appears to be one of the most promising. It is the purpose of this chapter to show how a simple algorithm for evaluating the superposition integral (Duhamel's integral) can provide a powerful tool for the analysis of complicated electrochemical systems not amenable to analysis by analytic methods. Software for use both with planar electrodes and with rotating disks is developed, and the method is applied to the analysis of cyclic voltammograms for metal-ion deposition and iodide-ion oxidation.

3.3. Application of Duhamel's Superposition Integral to Linear Diffusion

In simple diffusion, the mass transfer of a species i in solution is determined by Fick's second law:

$$\frac{\partial c_i}{\partial t} = D_i \frac{\partial^2 c_i}{\partial x^2} . \tag{3-1}$$

Since this equation is linear, the flux to the electrode surface resulting from any arbitrary (even nonlinear) surface boundary conditions can be obtained by superposing the flux contributions resulting from simple step changes in concentration at the surface. An illustration of the principle is shown in figure 3-2. Each small step change is initiated at a different time, and the effect of that step at future times is damped as the response decays. In the limit of an infinite number of small surface-concentration changes, the flux resulting from the combination of infinitesimal step

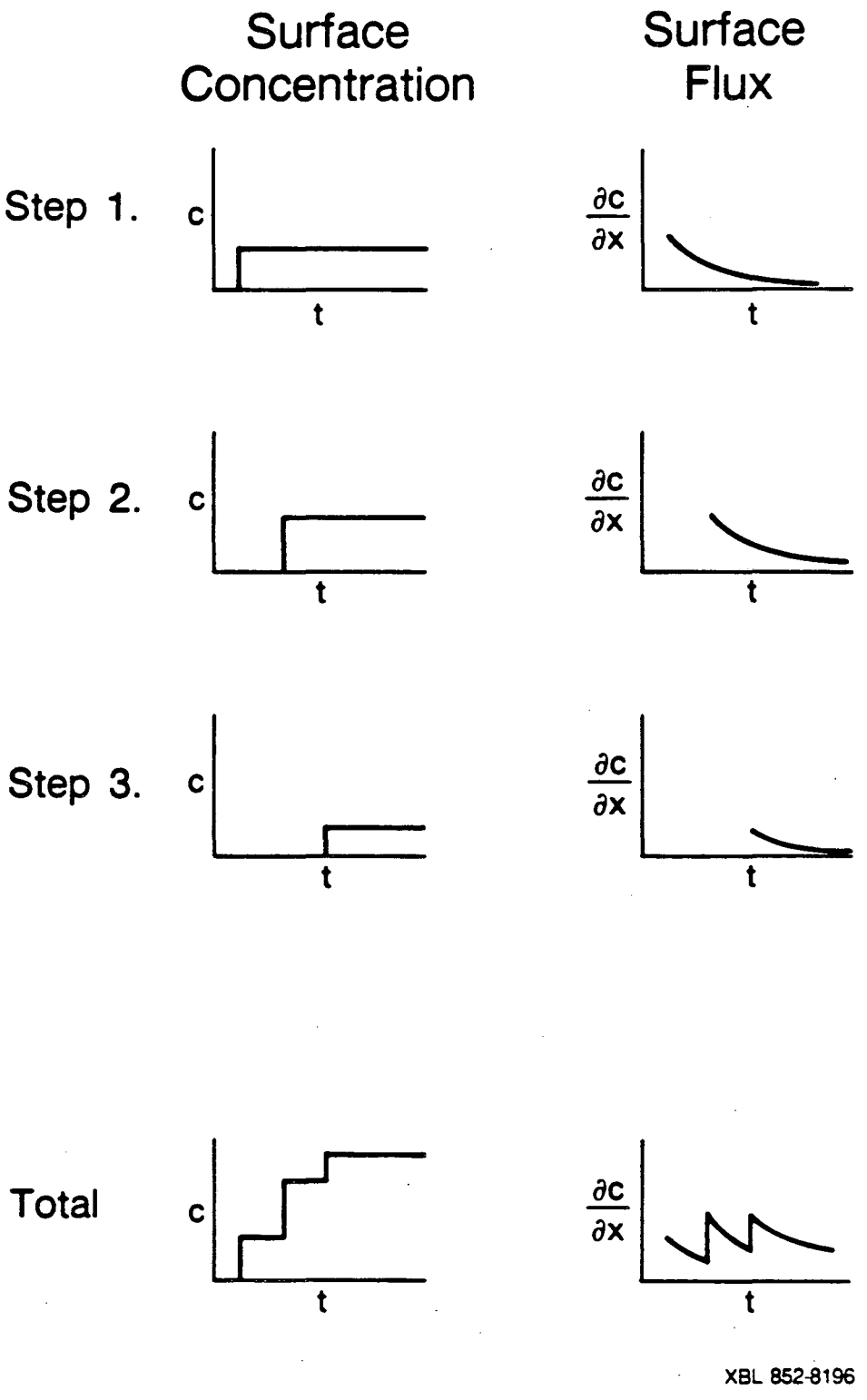


Figure 3-2. Illustration of the principle of superposition.

changes is represented by the integral, known as Duhamel's integral [6], shown below:†

$$\frac{\partial c_i}{\partial x}(0,t) = \int_0^t \frac{\partial c_i}{\partial t}(0,\tau) \cdot \frac{\partial \overline{c}_i}{\partial x}(0,t-\tau)\, d\tau . \tag{3-2}$$

The utility of the approach is that the flux profiles resulting from complicated, nonlinear surface conditions can be determined by a simple integral equation involving the concentration field arising from the step change problem (for which an analytic solution is generally available). Specifically, the value of the surface flux, as a function of time, is calculated from the integral of the surface-flux response to a unit step change in surface concentration, multiplied by the time variation of the surface concentration. The time variation of the surface concentration is, in turn, dependent upon the potential wave form, through the influence of potential on the reactions at the electrode surface.

For machine calculation, the integral must be approximated by a finite number of discrete parts, and a method for performing this discretization has been suggested by Wagner [7] and by Acrivos and Chambré [8]:

First, the complete integral is divided into a sum of n smaller integrals:

$$\frac{\partial c_i}{\partial x}(0,t) = \sum_{k=0}^{n-1} \int_{k\Delta t}^{(k+1)\Delta t} \frac{\partial c_i}{\partial t}(0,\tau) \cdot \frac{\partial \overline{c}_i}{\partial x}(0,t-\tau)\, d\tau , \tag{3-3}$$

where Δt is the time-step size, k a summation index, and n the number of steps. Next, the integrals at each step are simplified by assuming that the surface concentration is a linear function of time over the interval $k\Delta t$ to $(k+1)\Delta t$. Thus, the slope, $\frac{\partial c_i}{\partial t}$, is taken to be constant over the interval k, and can be approximated as:

$$\frac{\partial c_i}{\partial t}(0,\tau) \approx \frac{c_{i,k+1} - c_{i,k}}{\Delta t} . \tag{3-4}$$

† The bar above the concentration indicates that it is the result of a unit-step change. ($\overline{c}_i$ is dimensionless.)

Finally, then, with this approximation, the sum can be rewritten:

$$\frac{\partial c_i}{\partial x}(0,t) = \sum_{k=0}^{n-1} \frac{c_{i,k+1}-c_{i,k}}{\Delta t} \cdot \int_{k\Delta t}^{(k+1)\Delta t} \frac{\partial \overline{c}_i}{\partial x}(0,t-\tau)\, d\tau , \tag{3-5}$$

and this is the form suggested in [8] for performing the calculation mechanically.

3.4. An Algorithm for Coupled Systems Based on Duhamel's Integral

The expression above for computing the superposition integral can be used as the basis for a general algorithm for calculating the surface concentrations and surface fluxes for an arbitrary number of species in an electrochemical system. The procedure is complicated, since at the surface of the electrode ($x = 0$) a set of coupled nonlinear functions of the surface concentrations and the surface fluxes of all the species must be satisfied. At a step n (corresponding to time $t = n\,\Delta t$), the values of both the surface concentrations and surface fluxes are unknown and must be determined.

The number of equations and unknowns can be reduced, since, for any trial values of surface concentrations, equation (3-5) allows the surface flux to be calculated directly. Thus, by combining Duhamel's integral (3-5) with the surface boundary conditions a smaller set of equations containing only the surface concentrations can be created.

To solve this set of equations, a multi-dimensional Newton-Raphson procedure can be used. In this procedure, the nonlinear equations are linearized in a Taylor series expansion about a set of trial values for the surface concentrations, where the derivatives needed for the expansion are obtained from the equations by numerical differentiation in much the same way as in the BandAid technique.† New values of the concentrations are then obtained from the linearized equations by a simple matrix-inversion technique, and the resulting concentrations are used as new trial values. This procedure is repeated until the resulting concentrations no longer differ from the trial values.

† See Part 2 of this dissertation for details.

Once the surface concentrations have been obtained in this way, the surface fluxes resulting from those concentrations are computed by (3-5). Finally, the total current density to the electrode is obtained by adding the contribution to the current from the surface flux of each species, and a simulated voltammogram is produced.

3.4.1. Simplifications of the Integral Calculations

Two major simplifications to the calculation of Duhamel's integral from (3-5) can be made, and these modifications, outlined below, reduce considerably the computational requirements for the algorithm.

First, a large increase in the efficiency of the calculation can be made by observing that, at a step n, the values of the surface concentrations for all steps up to $n-1$ have already been calculated. Hence, it is possible to evaluate all of the terms in the summation except the last prior to any iteration at all. This can be seen by splitting the sum into two parts, a known constant part (for $k = 0$ to $n-2$) and an unknown part consisting of a single term. Thus, (3-5) may be rewritten:

$$\frac{\partial c_i}{\partial x}(0,t) = \sum_{k=0}^{n-2} \frac{c_{i,k+1}-c_{i,k}}{\Delta t} \cdot \int_{k\Delta t}^{(k+1)\Delta t} \frac{\partial \bar{c}_i}{\partial x}(0,n\,\Delta t-\tau)\,d\tau$$

$$+ \frac{c_{i,n}-c_{i,n-1}}{\Delta t} \cdot \int_{(n-1)\Delta t}^{n\Delta t} \frac{\partial \bar{c}_i}{dx}(0,n\,\Delta t-\tau)\,d\tau \quad . \tag{3-6}$$

A second, additional simplification is possible, because each integral $\int_{k\Delta t}^{(k+1)\Delta t} \frac{\partial \bar{c}_i}{\partial x}(0,n\,\Delta t-\tau)\,d\tau$ depends only upon the surface flux from the unit-step problem and its value is a function only of the index $n-k$. All of the integrals in (3-6), therefore, are constants and need to be calculated only once (as a function of $n-k$) and stored. These constant integrals may be represented by the symbol $A_{i,n-k}$ and the series rewritten more compactly as

$$\frac{\partial c_i}{\partial x}(0,t) = \sum_{k=0}^{n-2} \frac{c_{i,k+1}-c_{i,k}}{\Delta t} \cdot A_{i,n-k} + \frac{c_{i,n}-c_{i,n-1}}{\Delta t} \cdot A_{i,1} , \tag{3-7}$$

where

$$A_{i,n-k} = \int_{k\Delta t}^{(k+1)\Delta t} \frac{\partial \bar{c}_i}{\partial z}(0, n\,\Delta t - \tau)\, d\tau , \tag{3-8}$$

For calculational convenience, the expression for $A_{i,n-k}$ can be expressed as a difference of two simpler integrals:

$$A_{i,n-k} = a_i[\,(n-k)\Delta t\,] - a_i[\,(n-k-1)\Delta t\,] , \tag{3-9}$$

where

$$a_i(t) = \int_0^t \frac{\partial \bar{c}_i}{\partial x}(0,\tau)\, d\tau . \tag{3-10}$$

With these modifications, a quick algorithm for calculating the fluxes and surface concentrations as a function of time can be developed, and it is outlined in figure 3-3. To illustrate the implementation of the algorithm, expressions $a_i(t)$ for two electrode geometries are presented: diffusion to a planar electrode in stagnant solution (semi-infinite stagnant diffusion), and diffusion to a planar electrode adjacent to a Nernst stagnant diffusion layer. In each case, the functions $a_i(t)$ are different, since each corresponds to the solution of a slightly different unit-step-change problem. All other aspects of the numerical integration, however, remain unchanged.

Figure 3-3. Outline of the Computational Algorithm

---

begin

evaluate and store the $A_{i,n-k}$

for n := 1 **to** n_{max} **do**
begin

compute and store $\sum_{k=0}^{n-2} \frac{c_{i,k+1}-c_{i,k}}{\Delta t} \cdot A_{i,n-k}$

use Newton-Raphson to solve for the $c_{i,n}$ (iterative)

determine $\frac{\partial c_i}{\partial x}$ from the complete sum (3-5)

calculate the current density from the $\frac{\partial c_i}{\partial x}$ by Faraday's law

end

end

3.5. Expressions for Semi-infinite Stagnant Diffusion

The unit-step problem to be solved in this case is the following:

$$\frac{\partial \bar{c}_i}{\partial t} = D_i \frac{\partial^2 \bar{c}_i}{\partial x^2} \tag{3-11}$$

with initial and boundary conditions

$$\text{at} \quad t = 0, \qquad \bar{c}_i = 0, \tag{3-12a}$$

$$\text{at} \quad x = 0, \qquad \bar{c}_i = 1, \tag{3-12b}$$

and

$$\text{at} \quad x = \infty, \qquad \bar{c}_i = 0. \tag{3-12c}$$

The problem can be solved by Laplace transformation or by similarity transformation, with the result:

$$\overline{c}_i = 1 - \frac{2}{\sqrt{\pi}} \int_0^{\eta} \exp(-\eta^2) d\eta , \tag{3-13}$$

where the similarity variable, η, is defined as

$$\eta = \frac{x}{\sqrt{4 D_i t}} . \tag{3-14}$$

The surface derivative, $\frac{\partial \overline{c}_i}{\partial x}(0,t)$, is determined by differentiation:

$$\frac{\partial \overline{c}_i}{\partial x}(0,t) = \frac{\partial \overline{c}_i}{\partial \eta} \frac{\partial \eta}{\partial x} = \frac{-1}{\sqrt{\pi D_i t}} , \tag{3-15}$$

and the integrals $a_i(t)$ can be determined from (3-10) as,

$$a_i(t) = -2 \sqrt{t/\pi D_i} \tag{3-16}$$

3.6. Expressions for a Nernst Stagnant Diffusion Layer

The unit-step problem to be solved here is very similar to that in the previous case, the only difference being the boundary condition away from the surface. Thus,

$$\frac{\partial \overline{c}_i}{\partial t} = D_i \frac{\partial^2 \overline{c}_i}{\partial x^2} \tag{3-17}$$

with initial and boundary conditions

$$\text{at} \quad t = 0 , \qquad \overline{c}_i = 0 , \tag{3-18a}$$

$$\text{at} \quad x = 0 , \qquad \overline{c}_i = 1 , \tag{3-18b}$$

and

$$\text{at} \quad x = \delta_i , \qquad \overline{c}_i = 0 . \tag{3-18c}$$

The general result for the concentration field can be written in terms of a steady-state solution and a Fourier-series expansion for the transient terms as follows [9]:

$$\overline{c}_i = 1 - \frac{x}{\delta_i} - \sum_{m=1}^{\infty} \frac{2}{m\pi} \sin\left(\frac{m\pi}{\delta_i} x\right) \exp\left(-\frac{m^2\pi^2 D_i}{\delta_i^2} t\right) . \tag{3-19}$$

By differentiation,

$$\frac{\partial \overline{c}_i}{\partial x}(0,t) = \frac{-1}{\delta_i} - \sum_{m=1}^{\infty} \frac{2}{\delta_i} \exp\left(-\frac{m^2\pi^2 D_i}{\delta_i^2} t\right) , \tag{3-20}$$

and the integral $a_i(t)$ is found to be

$$a_i(t) = \frac{-t}{\delta_i} - \sum_{m=1}^{\infty} \frac{2\delta_i}{D_i m^2\pi^2} \exp\left(-\frac{m^2\pi^2 D_i}{\delta_i^2} t\right) . \tag{3-21}$$

Notice that the second term in the expression for $a_i(t)$ consists of an infinite series expansion. For numerical evaluation, the infinite series is approximated by a finite number of terms as

$$a_i(t) = \frac{-t}{\delta_i} - \sum_{m=1}^{m_{max}} \frac{2\delta_i}{D_i m^2 \pi^2} \exp\left(-\frac{m^2 \pi^2 D_i}{\delta_i^2} t \right) , \qquad (3\text{-}22)$$

where only the first m_{max} terms in the series are retained. Although the series is convergent, at short times ($t \leq \frac{4\delta_i^2}{\pi D_i}$) the small number of terms m_{max} is not sufficient to calculate $a_i(t)$ to the accuracy necessary for the simulation. To gain greater accuracy, the short-time behavior is represented by the solution for semi-infinite stagnant diffusion shown in the previous section. Thus, the complete solution for the Nernst stagnant diffusion layer consists of two parts:

$$a_i(t) = \begin{cases} -2\sqrt{t/\pi D_i} & \text{for } t \leq \frac{4\delta_i^2}{\pi D_i} \\ \frac{-t}{\delta_i} - \sum_{m=1}^{m_{max}} \frac{2\delta_i}{m^2 \pi^2 D_i} \exp\left(-\frac{m^2 \pi^2 D_i}{\delta_i^2} t \right) & \text{for } t > \frac{4\delta_i^2}{\pi D_i} , \end{cases} \qquad (3\text{-}23)$$

which produce an accurate representation of $a_i(t)$ for all times.

3.7. Expressions for Other Electrode Systems

To extend the analysis shown here to other electrode systems, the solution to the unit-step problem for the other systems must be known. From that solution, an expression for $a_i(t)$ can be determined, and the $A_{i,n-k}$ obtained. Once the $A_{i,n-k}$ have been found, however, the algorithm does not differ in any way from the outline in figure 3-3.

For example, a better treatment of the rotating-disk system can be made without the assumption of a Nernst stagnant diffusion layer, by solving the complete convective-diffusion equation with the velocity determined from the fluid motion to the disk electrode. Such an analysis has been made by Nisancioglu and Newman [10], and more accurate simulations can be obtained if the results of their work are used in place of the Nernst model shown here. Although for simplicity the discussion here is restricted to the stagnant-diffusion-layer case, use of the more rigorous expressions in [10] would not require any additional computational effort.

3.8. Boundary Conditions for the Electrode Surface during Cyclic Voltammetry

The governing differential equation for mass-transfer of a species i to an electrode surface in the absence of migration, convection, and homogeneous chemical reactions is simply Fick's second law:

$$\frac{\partial c_i}{\partial t} = D_i \frac{\partial^2 c_i}{\partial x^2} , \tag{3-24}$$

and the numerical algorithm illustrated in figure 3-3 can be used directly, provided that boundary conditions on the electrode surface can be supplied. In this section, boundary conditions suitable for simulating cyclic voltammograms of typical electrochemical systems are developed.

Consider a system of N reacting species, which can react in M simultaneous reactions at the electrode surface. The flux N_i of each species i to the electrode surface is proportional to the concentration derivative, $\frac{\partial c_i}{\partial x}$, and it is equal to the rate of reaction, R_i. Thus, the boundary conditions consist of a set of N equations:

$$N_i = R_i , \tag{3-25}$$

where

$$N_i = -D_i \frac{\partial c_i}{\partial x} , \tag{3-26}$$

and

$$R_i = \sum_{j=1}^{M} \frac{s_{i,j} i_j}{n_j F} . \tag{3-27}$$

The current density i_j for each reaction j is related to the surface concentrations and applied cell potential by a generalized kinetic expression, the Butler-Volmer expression:

$$i_j = i_{oj,ref} \left[\prod_i \left(\frac{c_i}{c_{ij,ref}} \right)^{p_{i,j}} \exp(\alpha_{a,j} \eta) - \prod_i \left(\frac{c_i}{c_{ij,ref}} \right)^{q_{i,j}} \exp(-\alpha_{c,j} \eta) \right] , \tag{3-28}$$

where

$$p_{i,j} = \begin{cases} s_{i,j} & \text{if } s_{i,j} > 0 \\ 0 & \text{if } s_{i,j} \leq 0 \end{cases},$$

$$q_{i,j} = \begin{cases} -s_{i,j} & \text{if } s_{i,j} < 0 \\ 0 & \text{if } s_{i,j} \geq 0 \end{cases},$$

$$\eta = \frac{F}{RT}(V - \Phi_{ref} - \Phi_{ohm} - U_{j,ref}),$$

and the applied potential, $V - \Phi_{ref}$, is swept linearly at a rate of b volts per second.

For an anodic sweep:

$$V - \Phi_{ref} = V_{\min} + bt_a,$$

and for a cathodic sweep:

$$V - \Phi_{ref} = V_{\max} - bt_c,$$

where t_a and t_c are times into an anodic or cathodic sweep, respectively.

For each reaction j, the exchange-current density, $i_{oj,ref}$, is based on reference concentrations $c_{ij,ref}$, and the potential $U_{j,ref}$ is related to the standard cell potential U_j^ℓ according to the Nernst equation:

$$U_{j,ref} = U_j^\ell - \sum_{i=1}^{N} \frac{s_{i,j} RT}{n_j F} \ln\left(\frac{c_{ij,ref}}{\rho_o}\right) - U_{re}. \qquad (3\text{-}29)$$

where ρ_o is the solvent density and U_{re} is the potential of the reference electrode.

The ohmic potential drop, Φ_{ohm}, is estimated from the primary resistance to a disk electrode [11]:

$$\Phi_{ohm} = \frac{\pi r_o i}{4\kappa}, \qquad (3\text{-}30)$$

where the current density to the disk is computed from the flux of each species:

$$i = \sum_{i=1}^{N} F z_i N_i . \tag{3-31}$$

Since the ohmic drop depends upon the current, both the overpotential, η, and the current density, i, are unknown until convergence is achieved at each step.

At a rotating disk electrode, the equivalent Nernst stagnant diffusion layer thickness is obtained from the rotation speed as follows:†

$$\delta_i = 1.61 D_i^{1/3} \Omega^{-1/2} \nu^{1/6} , \tag{3-32}$$

and an equivalent mass-transfer coefficient can be defined

$$k_{m,i} = \frac{D_i}{\delta_i} . \tag{3-33}$$

When the rotation speed is zero, the steady-state boundary-layer thickness tends toward infinity, and the coefficient $k_{m,i}$ is zero. In that case, the flux resulting from a step-change at an electrode surface in a semi-infinite medium is used in the superposition integral. For cases of non-zero $k_{m,i}$, the flux for the Nernst stagnant diffusion layer can be used.

The problem specification is completed by setting the initial ($t = 0$) concentration of each species equal to its bulk value $c_{i,o}$ everywhere in solution.

3.9. Development of a Software Package

The formulas derived in the previous sections can be used to develop computer programs for predicting the transient current response of electrochemical systems, and a general subprogram for this purpose is contained in the appendix. It consists of two parts: a specific procedure (SuperPose) for calculating the superposition integrals with the algorithm outlined in figure 3-3 and a general multi-dimensional Newton-Raphson procedure (NewtRaph), which is called by SuperPose to perform the iterative loop. To use SuperPose and NewtRaph for simulation of the response an

† Note that the diffusion-layer thicknesses δ_i are not identical for every species i when the diffusion coefficients D_i are unequal.

electrochemical system, a main calling program containing a function subprogram that describes the surface boundary conditions must be supplied. The organization of the subprogram hierarchy is shown in figure 3-4.

The form of the function subprogram describing the problem is similar to the form used for programs calling the BandAid package (see Part 2 of this dissertation). The subprogram contains expressions for the boundary conditions as functions of the surface concentrations, the surface fluxes, and time, and these expressions normally include relations for the electrode kinetics as well as a description of the wave form of the applied potential (as a function of time). The functional description can employ any wave forms and kinetic expressions, but the application illustrated here is restricted to triangular waveforms and generalized Butler-Volmer kinetics for the surface reactions (*i. e.*, the model developed in the previous section). Other applications programs (for example, pulse waveforms with adsorption kinetics) can be created by appropriate modification of the function subprogram.

CycVolt, a complete calling program including the boundary conditions for the model in section 3.8 has been combined with subprograms SuperPose and NewtRaph to produce a software package for the simulation of cyclic voltammograms. With this program, voltammograms for a large number of electrochemical systems can be simulated, and two applications are shown here to illustrate its use. To run CycVolt, a user must supply a time-step size and a limit on the number of iterations to be attempted by the Newton-Raphson routine, as well as physical-property data, kinetic constants, and operating conditions. A complete list of the quantities needed for a simulation is shown in tables accompanying the examples.

3.10. Linear Sweep Voltammograms for Mercury Deposition

This first example simulates linear sweep voltammograms for mercury deposition at a glassy-carbon rotating disk electrode. Linear sweep voltammetry is a special case of cyclic voltammetry, wherein the wave form, rather than a triangular wave, is simply a ramp function.

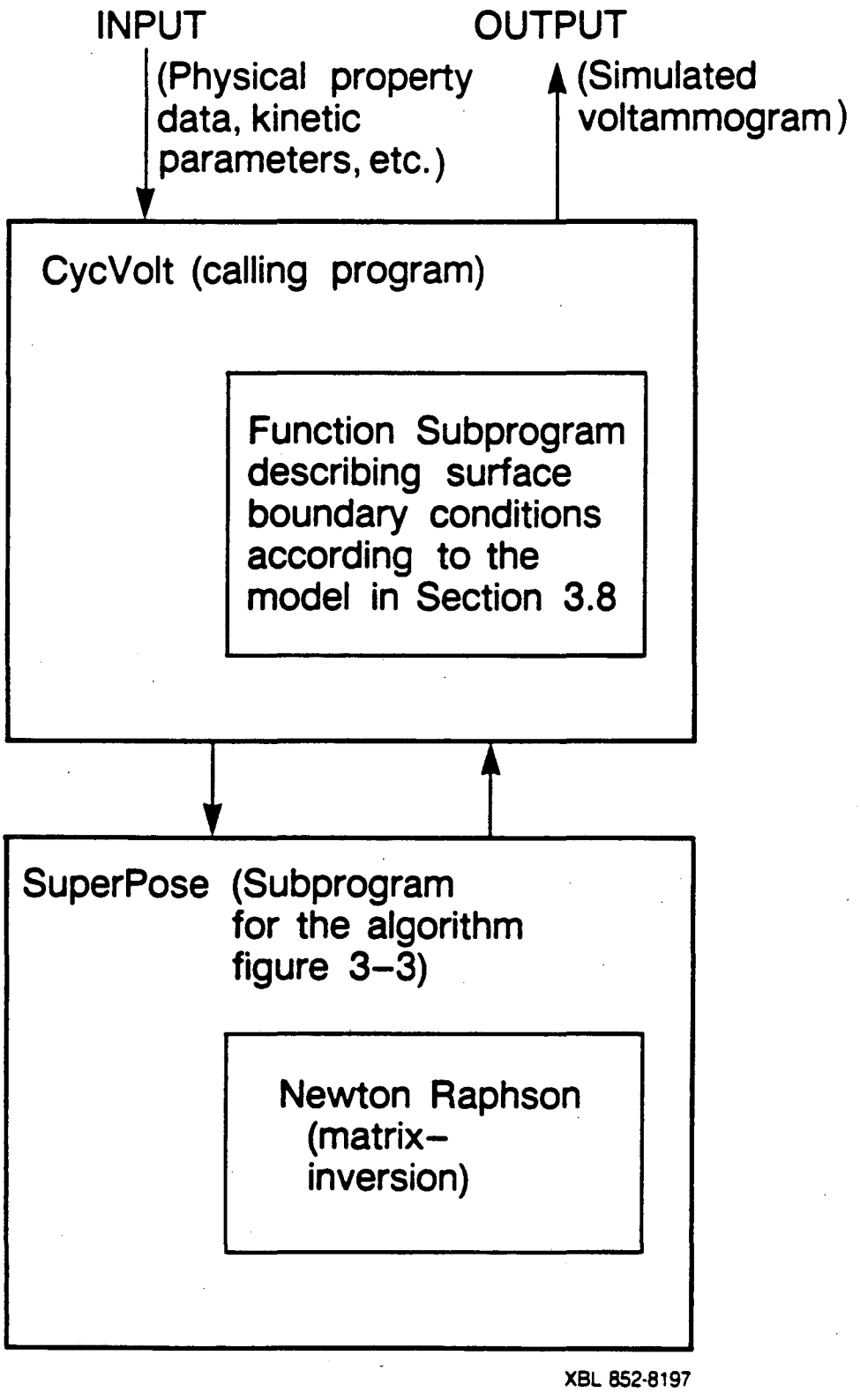


Figure 3-4. Organization of subprograms for calculating of Duhamel's integral.

The mercury reduction reaction, discussed in more detail in Part 1 of this dissertation, is shown below:

$$HgCl_4^{-2} + 2e^- \rightarrow Hg(l) + 4Cl^- .$$

In concentrated chloride solution (here, 4 M NaCl), the mercuric ion, Hg^{+2}, is complexed with chloride as $HgCl_4^{-2}$. Physical property data and parameters for the model are shown in table 3-1.

Table 3-1. Values of Parameters for Model Calculations

$N = 2$	$M = 1$
Iteration Limit = 10	Convergence Tolerance = 1×10^{-8}
ΔV (step size) = 0.025 V	
$s_{HgCl_4^{-2}} = -1$	$s_{Cl^-} = 4$
$D_{HgCl_4^{-2}} = 1.5\times10^{-5}$ cm^2/s	$D_{Cl^-} = 2.032\times10^{-5}$ cm^2/s
$z_{HgCl_4^{-2}} = -2$	$z_{Cl^-} = -1$
$c_{HgCl_4^{-2},ref} = 5\times10^{-7}$ mol /cm^3	$c_{Cl^-,ref} = 4\times10^{-3}$ mol /cm^3
$c_{HgCl_4^{-2},o} = 5\times10^{-7}$ mol /cm^3	$c_{Cl^-,o} = 4\times10^{-3}$ mol /cm^3
$T = 298.15$ K	$\rho_o = 1.14\times10^{-3}$ kg /cm^3
$\nu = 0.01$ cm^2/s	$\kappa = 0.199$ mho/cm
$U_{re} = 0.241$ V	$U^\theta = 0.4138$ V
$n = 2$	$i_{o,ref} = 1.5\times10^{-3}$ A /cm^2
$\alpha_a = 1.4$	$\alpha_c = 0.6$
$V_{min} = -0.6$ V	$V_{max} = 0.0$ V
$b = 5.3\times10^{-4}$ V /s	$r_o = 0.375$ cm
Rotation Speed = 2500 rpm	$\Omega = 261.8$ rad/s
$\Delta t = 47.17$ s	$U_{ref} = 0.0276$ V
$p_{HgCl_4^{-2}} = 0$	$p_{Cl^-} = 4$
$q_{HgCl_4^{-2}} = 1$	$q_{Cl^-} = 0$
$\delta_{HgCl_4^{-2}} = 0.00114$ cm	$\delta_{Cl^-} = 0.00126$ cm
$k_{m,HgCl_4^{-2}} = 0.013169$ cm/s	$k_{m,Cl^-} = 0.0161$ cm/s
$A = 0.4418$ cm^2	

and model voltammograms (compared to experiment at a number of rotation speeds) are shown in figure 3-5.† Although only a single reaction is involved in this system, the fluxes of both $HgCl_4^{-2}$ and Cl^- ions must be calculated to determine the current.

As an alternative mechanism (since the chloride ion is present in large excess and the diffusion of the mercuric chloride complex limits the current), the system behavior may be approximated by assuming that the mercuric ion is present as Hg^{+2} and that the effective electrochemical reaction is

$$Hg^{+2} + 2e^- \rightarrow Hg(l) .$$

Note, however, that if this is done, the value of U^0 for the reduction of the complexed species (adjusted for the Cl^- concentration) must be used in the calculation.

The modified mechanism is simpler than the original problem, since it involves only a single electroactive species, and analytic solutions to the mass-transfer problem are available for some cases. Andricacos and Ross [12], for example, have developed analytic solutions for metal-deposition problems of this type for a single electroactive species when the electrode kinetics are very fast (reversible reactions). Although these results are helpful as guides for the numerical work, as the number of species and reactions becomes larger (or as electrode kinetics become important), the interactions between the species become too complicated for analytic solutions to be used directly. The second example application, shown below, illustrates more clearly the need for numerical simulation in these complicated cases.

† Since the sweep rate is slow (0.5 mV/s), the system is essentially in a quasi-steady state. For higher sweep rates, the voltammograms overshoot the limiting current before leveling off, and the overshoot is a function of sweep rate. (For a discussion of this phenomenon, see [12]).

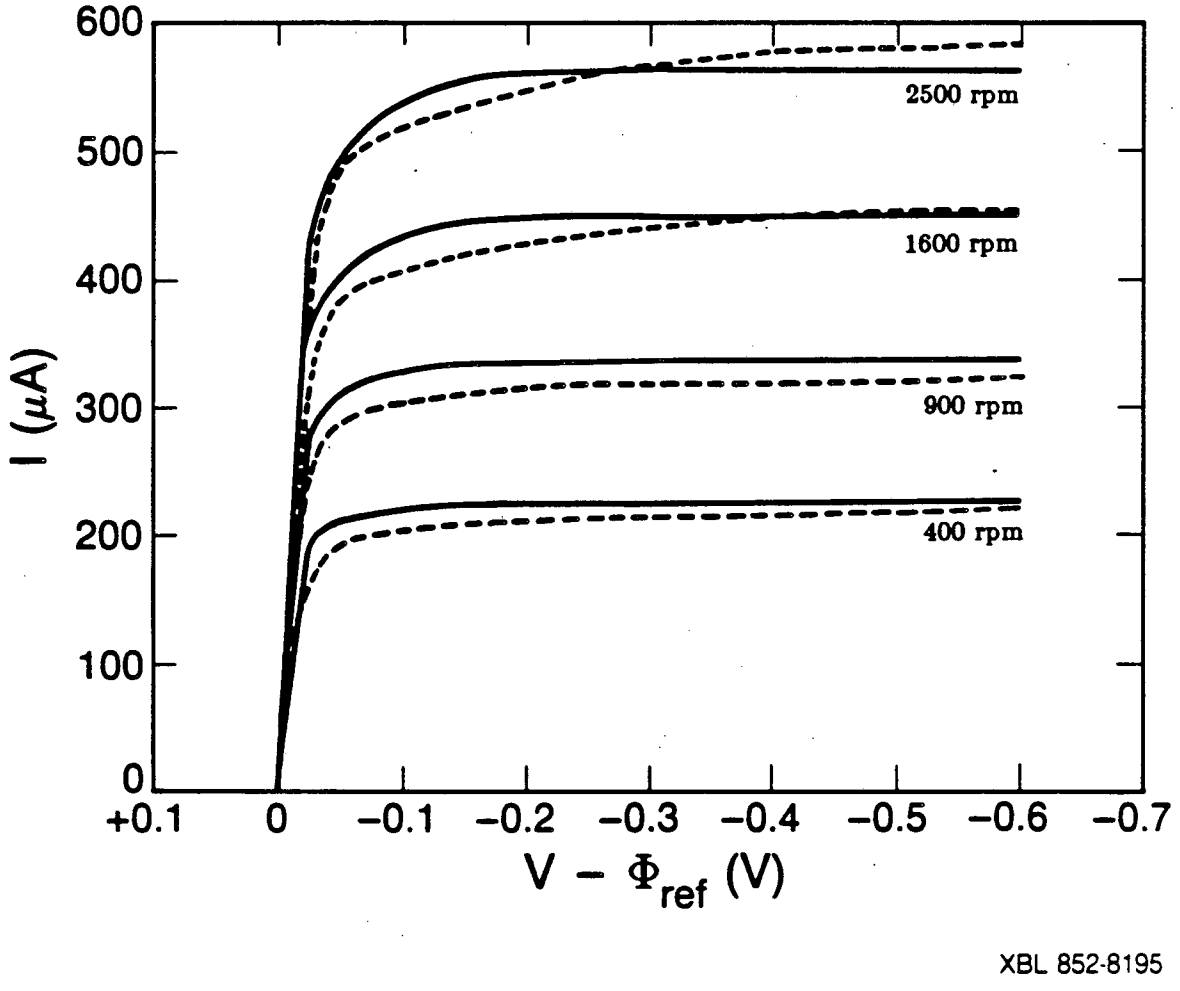


Figure 3-5. Model voltammograms for mercury deposition at a rotating disk electrode compared to experiment. Dashed curves show the experiment and solid curves show the simulation.

3.11. Cyclic Voltammograms for Iodide Redox Reactions in Propylene Carbonate

The cyclic voltammetry of the iodide-triiodide-iodine redox system in propylene carbonate is more complicated than that for simple metal deposition. The reaction stoichiometries are non-unity (3:1 and 2:3), and the reaction of each species influences the rate of reaction of the others. In principle, however, the mathematical model for this system is the same as that for the simple metal-ion deposition, and the approach is sufficiently general to allow both problems to be treated in essentially the same manner. No modification of the model is required, and the same computer program is used. Notice that, unlike the first example, the electrode in this study is stationary, and, therefore, the superposition integrals for semi-infinite stagnant diffusion are used for all calculations by the program. Parameters for the model calculations are shown in table 3-2.

Hanson *et al.* [13] have postulated the following mechanism for the oxidation of iodide ion at a platinum electrode in propylene carbonate solution:

$$3I^- \rightarrow I_3^- + 2e^- \qquad (I)$$

$$2I_3^- \rightarrow 3I_2 + 2e^- \qquad (II).$$

According to the mechanism, the iodide ion is first oxidized to triiodide ion (which is considerably more stable in propylene carbonate than in water). Then, as the applied potential is increased, the triiodide is further oxidized to iodine. This mechanism can be checked by comparing model voltammograms to experiment, and this is shown in figure 3-8.

Reaction (I), iodide oxidation, appears to be irreversible, since the peak on the reverse (cathodic) wave is shifted far to the left, whereas reaction (II), triiodide oxidation, exhibits essentially reversible (equilibrium) behavior. These characteristics are simulated in the model by choosing a much higher exchange-current density for reaction (II) than for reaction (I). In this way, kinetic resistances for reaction (II) are eliminated, while the irreversible behavior of reaction (I) is retained. In principle, a special-purpose model containing a Butler-Volmer expression for reaction (I) and the Nernst equilibrium expression for reaction (II) can be developed, and, in fact, this has been done [4,13]. It is not necessary, however, since, for high exchange-current densities,

Table 3-2. Values of Parameters for Model Calculations

$N = 3$	$M = 2$
Iteration Limit = 10	Convergence Tolerance = 1×10^{-8}
ΔV (step size) = 0.025 V	
$s_{I^-,I} = 3$	$s_{I^-,II} = 0$
$s_{I_2,I} = 0$	$s_{I_2,II} = -3$
$s_{I_3^-,I} = -1$	$s_{I_3^-,II} = 2$
$D_{I^-} = 2.0 \times 10^{-6}$ cm^2/s	$z_{I^-} = -1$
$D_{I_2} = 2.0 \times 10^{-6}$ cm^2/s	$z_{I_2} = 0$
$D_{I_3^-} = 2.0 \times 10^{-6}$ cm^2/s	$z_{I_3^-} = -1$
$c_{I^-,(I)ref} = 2.5 \times 10^{-6}$ mol /cm^3	$c_{I^-,(II)ref} = 2.5 \times 10^{-6}$ mol /cm^3
$c_{I_2,(I)ref} = 2.5 \times 10^{-6}$ mol /cm^3	$c_{I_2,(II)ref} = 2.5 \times 10^{-6}$ mol /cm^3
$c_{I_3^-,(I)ref} = 2.5 \times 10^{-6}$ mol /cm^3	$c_{I_3^-,(II)ref} = 2.5 \times 10^{-6}$ mol /cm^3
$c_{I^-,o} = 2.5 \times 10^{-6}$ mol /cm^3	$c_{I_2,o} = 8.8 \times 10^{-23}$ mol /cm^3
	$c_{I_3^-,o} = 1.1 \times 10^{-17}$ mol /cm^3
$T = 298.15$ K	$\rho_o = 1.20 \times 10^{-3}$ kg /cm^3
$\nu = 0.01$ cm^2/s	$\kappa = 5 \times 10^{-3}$ mho/cm
$U_{re} = 0.0$ V	
$U_I^\theta = 0.0$ V	$U_I^\theta = 0.6$ V
$n_I = 2$	$n_{II} = 2$
$i_{o,(I)ref} = 6.5 \times 10^{-5}$ A /cm^2	$i_{o,(II)ref} = 1.0 \times 10^{5}$ A /cm^2
$\alpha_{a(I)} = 1.6$	$\alpha_{a(II)} = 1.0$
$\alpha_{c(I)} = 0.4$	$\alpha_{c(II)} = 1.0$
$V_{min} = 0.4$ V	$V_{max} = 1.8$ V
$b = 0.2$ V /s	$r_o = 0.135$ cm
Rotation Speed = 0 rpm	$\Omega = 0$ rad/s
$\Delta t = 0.05$ s	A = 0.4418 cm^2
$U_{(I)ref} = 0.7540$ V	$U_{(II)ref} = 1.188$ V
$p_{I^-,I} = 3$	$p_{I^-,II} = 0$
$p_{I_2,I} = 0$	$p_{I_2,II} = 0$
$p_{I_3^-,I} = 0$	$p_{I_3^-,II} = 2$
$q_{I^-,I} = 0$	$q_{I^-,II} = 0$
$q_{I_2,I} = 0$	$q_{I_2,II} = 3$
$q_{I_3^-,I} = 1$	$q_{I_3^-,II} = 0$

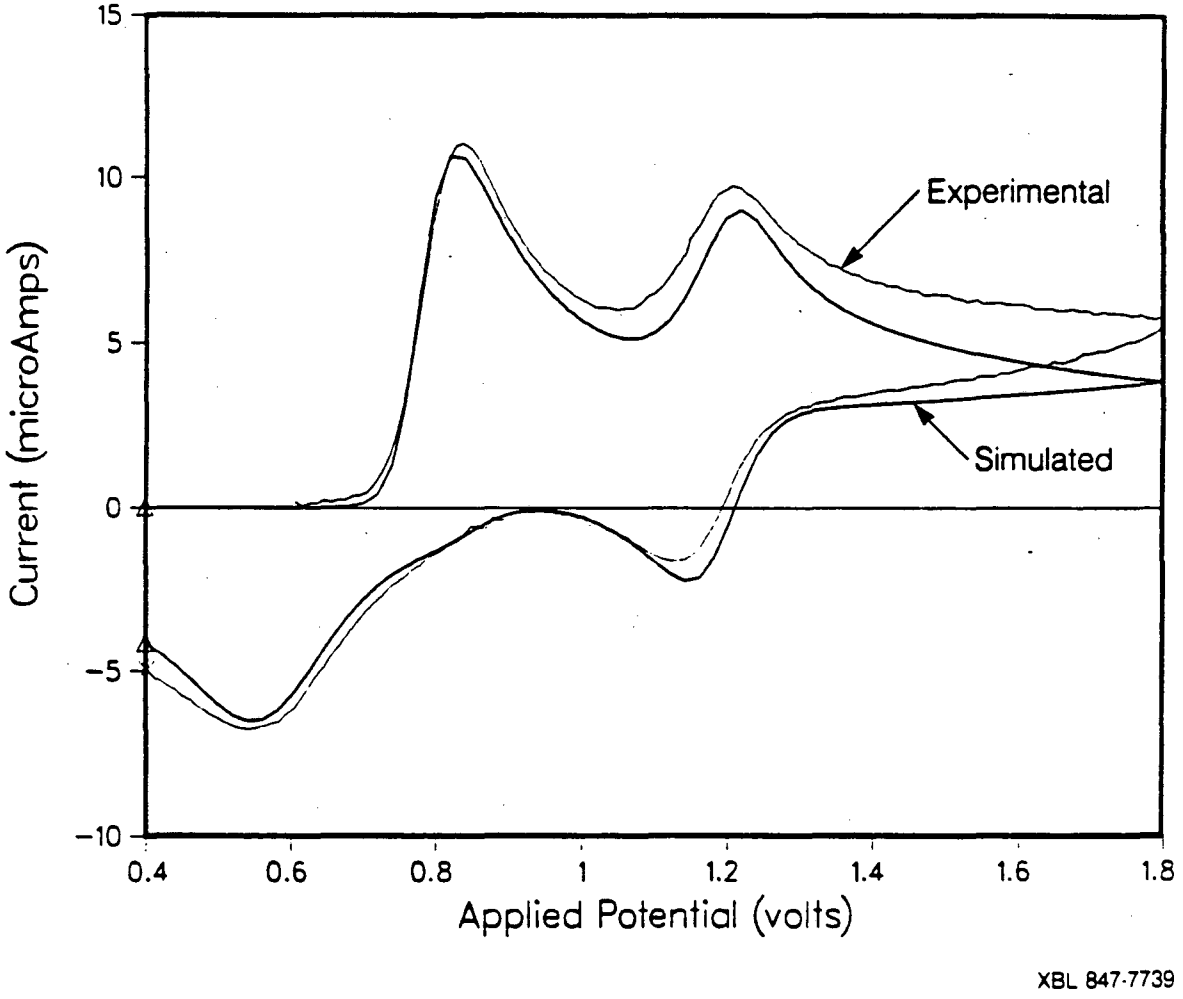


Figure 3-6. Comparison of model voltammograms to experiment. Parameter values and operating conditions are shown in table 3-2.

the Butler-Volmer expression reduces to the Nernst equation, and the reversible behavior can be simulated easily without altering the model equations.

To illustrate further the utility of the numerical simulation, figure 3-7 shows simulated voltammograms at a number of sweep rates. If the model truly represents the system, voltammograms from additional experiments at a variety of sweep rates should match these simulations. If they do not, the computer program may be used to test alternative reaction mechanisms until a more accurate representation of the behavior is found.

3.12. Conclusions and Significance

This work illustrates the power of superposition integrals for simulating the transient current response of electrochemical systems. The method developed here is the first step toward a comprehensive software package for the analysis not only of the transient current response from triangular waveforms but from other common waveforms as well. The two examples shown illustrate the use of the method and also point out the importance that modeling can play in the evaluation of complex reaction mechanisms. The numerical procedure is fast enough to be used with a microprocessor, and it should be possible to use the algorithm developed here as part of an on-line analysis system connected directly to an experimental data-acquisition system. With such a system, model predictions for any number of hypothesized reaction pathways could be compared directly to experimental measurement as soon as the data were obtained in the laboratory.

Acknowledgement

The author would like to express his thanks to Dr. Karrie Hanson for suggesting the modeling effort described here and for supplying experimental cyclic voltammograms.

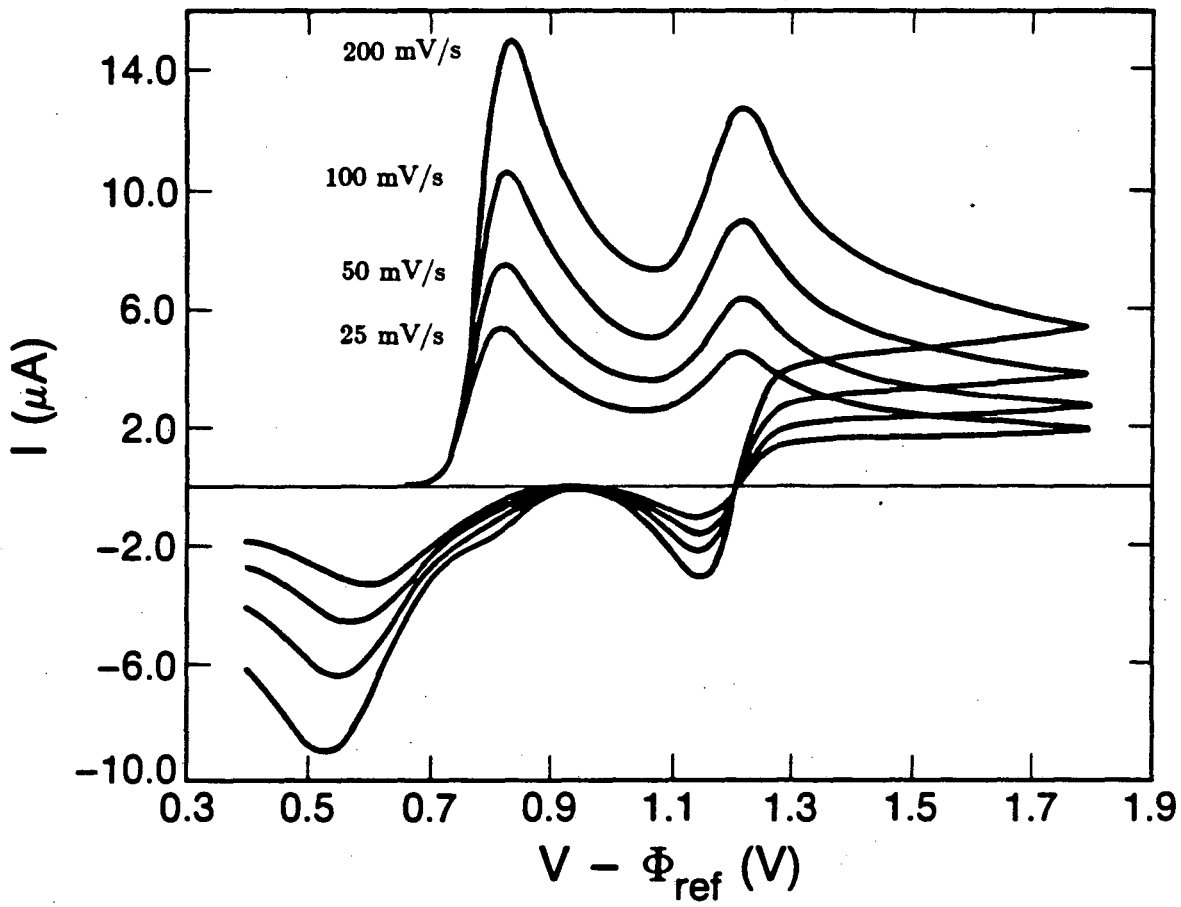


Figure 3-7. Simulated voltammograms as a function of sweeprate. Parameter values and all operating conditions but sweep rate are identical to those shown in table 3-2.

List of Symbols

a	Integral from $t = 0$ to t of the surface flux resulting from a step change in surface concentration at $t = 0$.
A_{n-k}	Coefficients in the discretized expression for Duhamel's integral.
b	Sweep rate, V/s
c_i	Concentration of species i, mol/cm^3
$\overline{c}_i$	Concentration of species i in the unit-step problem
$c_{ij,ref}$	Reference concentration of species i in reaction j, mol/cm^3
$c_{i,k}$	Concentration of species i at time-step k, mol/cm^3
$c_{i,n}$	Concentration of species i at time-step n, mol/cm^3
D_i	Diffusion coefficient of species i, cm^2/s
i	Index for species in solution
i	Current density at an electrode surface, A/cm^2
i_j	Partial current for reaction j, A/cm^2
$i_{oj,ref}$	Exchange-current density for reaction j at reference conditions, A/cm^2
j	Index for electrochemical reactions
k	Index for steps in the discrete form of the superposition integral
$k_{m,i}$	Effective film mass-transfer coefficient, cm/s
M	Number of simultaneous electrochemical reactions
m	Summation index in the Fourier series expansion for the transient terms in the expression for the flux resulting from a step change in concentration at a surface adjacent to a Nernst stagnant diffusion layer
m_{max}	Maximum number of terms used in the truncated Fourier series expansion
N	Number of species in solution

n	Number of steps in the discrete expression for the superposition integral
N_i	Flux of species i to the electrode surface, mol/cm^2-s
n_j	Number of electrons transferred in reaction j
n_{max}	Maximum number of steps in the expression for the superposition integral
$p_{i,j}$	Anodic reaction order for species i in reaction j
$q_{i,j}$	Cathodic reaction order for species i in reaction j
R	Universal gas constant, 8.314 J/mol-K
R_i	Reaction rate of species i at the electrode surface, mol/cm^2-s
r_o	Radius of the rotating disk electrode, cm
$s_{i,j}$	Stoichiometric coefficient of species i in reaction j (anodic reactant positive, cathodic reactant negative)
t	Time, s
Δt	Size of the time-step, s
t_a	Time into an anodic potential sweep, s
t_c	Time into a cathodic potential sweep, s
$U_{j,ref}$	Equilibrium potential of reaction j at the reference conditions relative to a given reference electrode, V
U_j^θ	Standard electrode potential of reaction j, V
U_{re}	Potential of reference electrode, V
$V - \Phi_{ref}$	Potential applied at the electrode surface relative to a reference electrode, V
V_{max}	Maximum applied potential, V
V_{min}	Minimum applied potential, V
x	Distance variable, cm
z_i	Valence of species i

$\alpha_{a,j}$	Anodic transfer coefficient for reaction j
$\alpha_{c,j}$	Cathodic transfer coefficient for reaction j
δ_i	Nernst stagnant diffusion-layer thickness for species i, cm
η	Total overpotential for electrochemical reaction (used in the Butler-Volmer expression for the rate of reaction)
η	Similarity variable used in the expression for the flux resulting from a step change in concentration at a surface in a semi-infinite stagnant medium
κ	Electrolyte conducivity, mho/cm
ν	Kinematic viscosity, cm^2/s
ρ_o	Solvent density, kg/cm^3
Φ_{ohm}	Ohmic potential drop, V
Φ_{ref}	Potential of reference electrode, V
τ	Variable of integration
Ω	Rotation speed of the rotating disk, rad/s

References

1. Allen J. Bard and Larry R. Faulkner, *Electrochemical Methods -- Fundamentals and Applications*, (New York: John Wiley & Sons, Inc., 1980).

2. Richard S. Nicholson and Irving Shain, "Theory of Stationary Electrode Polarography. Single Scan and Cyclic Methods Applied to Reversible, Irreversible, and Kinetic Systems," *Analytical Chemistry*, **36**, 706-723 (1964).

3. Mark S. Shuman, "Nonunity Electrode Reaction Orders and Stationary Electrode Polarography,", *Analytical Chemistry*, **41**, 142-146 (1969).

4. Karrie Hanson, *On the Electrochemistry of Halogen Electrodes in Propylene Carbonate*, dissertation, University of California, Berkeley (1985), LBL-18335.

5. Z. Galus, *Fundamentals of Electrochemical Analysis*, (London: John Wiley & Sons, Inc., 1976).

6. Francis B. Hildebrand, *Advanced Calculus for Applications*, (Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1976) pp. 463-467.

7. Carl Wagner, "On the Numerical Integration of Volterra Integral Equations," *Journal of Mathematical Physics*, **32**, 289-301 (1954).

8. Andreas Acrivos and Paul L. Chambré, "Laminar Boundary Layer Flows with Surface Reactions," *Industrial and Engineering Chemistry Fundamentals*, **49**, 1025-1029 (1957).

9. J. Crank, *The Mathematics of Diffusion*, second edition, (Oxford: Clarendon Press 1975), p. 50.

10. Kemal Nisancioglu and John Newman, "Transient Convective Diffusion to a Disk Electrode," *Journal of Electroanalytical Chemistry and Interfacial Electrochemistry*, **50**, 23-29 (1974).

11. John Newman, "Resistance for Flow of Current to a Disk," *Journal of the Electrochemical*

Society, **113** (1966), 501-502.

12. P. C. Andricacos and P. N. Ross, "Diffusion-Controlled Multisweep Cyclic Voltammetry I. Reversible Deposition on a Rotating Disk Electrode," *Journal of the Electrochemical Society*, **130**. 1340-1352 (1983).

13. Karrie Hanson, Michael Matlosz, John Newman, and Charles W. Tobias, "Theory of Cyclic Voltammetry for Reactions Having Complex Stoichiometry," presented at the 35th Meeting of the International Society of Electrochemistry, Berkeley, California, August 5-10, 1984.

Appendices

Appendix A-1: Modules in VAX-11 Pascal

Standard Pascal, as defined by the International Organization for Standardization (ISO), makes no allowance for the development of modules (*i. e.*, independently compiled units). According to the standard, all subprograms must be declared within the scope of the main program. Unfortunately, this is extremely inconvenient in the development of large programs. In the case here, the need to have the source listing of the BandAid routine nested within the user program creates two major problems. First, the BandAid code is long, and its inclusion in the main program unnecessarily obscures the readability of the user program. Second, the time needed to compile the BandAid procedure is generally much longer that that needed for the user program. Thus, there is a strong incentive to create BandAid as an independently compiled module or package. Although the standard makes no recommendation, virtually all Pascal implementations will have some facility for doing this. This document, in order to be specific, will use the facilities available in VAX-11 Pascal to create the modules.

In VAX-11 Pascal, a module containing object declarations is indicated by a first line containing the **module** statement (which is similar to the **program** statement of a normal Pascal program) and a final line containing the **end** statement:

```
module  ModName( input, output );

                .
                .
                .
          {... declarations ...}
                .
                .
                .
end
```

(Naturally, "ModName" can be any name chosen by the writer.)

A module may contain **const**, **type**, **var**, **function**, or **procedure** declarations, but it cannot contain a program body. Thus, although a module can be compiled, it cannot be executed independently. The objects declared in the module may, however, be used by a program (or by other modules). Compilation is the same as compilation of a normal Pascal program.

In order that the objects declared in the module may be referenced by other modules and programs, it is necessary to generate what is known as a Pascal environment file (.pen). This is accomplished by prefixing the ENVIRONMENT directive to the the module text as shown below:

```
[ENVIRONMENT('ModName.pen')]
```

Now, if a program (or another module) refers to objects declared in module "ModName", this is indicated at compile time by the INHERIT directive on the first line:

```
[INHERIT('ModName.pen')]
```

Thus, every program presented here will require an INHERIT statement to indicate that the program will make reference to AidMod, and any other program modules that may be used (in general, a module containing input/output procedures). Thus, for example, the CatReac program of the first example must be prefixed with the following directive:

```
[INHERIT('AidMod.pen','CatIO.pen')]
```

For the other example programs, the INHERIT statements are similar. Listings of the source code for all modules not shown in the text are presented later in the appendix.

To ensure as much portability as possible, the texts of the BandAid and BandShell procedures are written to conform to the Pascal standard. All non-standard features necessary for the implementation are contained in the AidMod module, which is

constructed around the source code. In this way, BandAid and BandShell should remain completely portable, thus facilitating transfers to machines other than VAXes.

The source code (in Standard Pascal) is inserted into the text of AidMod by use of the non-standard %INCLUDE directive of VAX-11 Pascal. The module is shown below, and the listings of BandShell, BandAid, and IOPkg procedures are shown in appendix B.

```
[ENVIRONMENT('AidMod.pen')]

module AidMod(input,output);

const    nLimit = 12;      WidthLimit = 25;
         ItLimit = 30;     jLimit = 1003;    {... Maximum Size Restrictions ...}

       SystemZero = 1e-30;
       SigDigits = 15;

type   RealNumber = double;
       ValueArray = array[1..nLimit, 1..jLimit]of RealNumber;
       DiffApprox = ( CenDiff, BackDiff, ForDiff,
                              Back3PtDiff, For3PtDiff );


         Vector = array [1..nLimit] of RealNumber;
         Matrix = array [1..nLimit] of Vector;

         IVector = array [1..nLimit] of integer;
         IMatrix = array [1..nLimit] of IVector;

         string = record chars : array [1..100] of char;
                         length : integer          end;


function SystemClock : integer;
begin  SystemClock := clock    end;


       %include 'BandShell.pas'
         %include 'BandAid.pas'

         %include 'IOPkg.pas'
         %include 'ListPrint.pas'



end  { module AidMod }
```

Appendix A-2: Command Files Necessary to Run the Example Programs

Shown below are two useful command files that can be used with version 4.1 of the VAX/VMS operating system to run the programs contained in this dissertation. It is assumed throughout that the files AidMod.pen and AidMod.obj (*i. e.*, the pascal environment file and object code for BandAid and BandShell), as well as all user programs (*e. g.*, CatReac.pas), are contained in directory [YOUR.DIRECTORY]. If they are stored in another directory, the appropriate directory name will need to be prefixed to those names when referring to the files. To be specific, all of the examples below feature the CatReac program, but the same steps can be used to run other programs as well.

Compiling a Program

The text of all Pascal programs should be stored in a file of type .pas (*e. g.* CatReac.pas). In general, the programs shown here consist of two parts: the main program and an input/output module. First, the input/output module should be compiled, since the main program will make reference to it. To compile an input/output module, the following command is issued:

```
$ pascal CatIO
```

This command generates object code, CatIO.obj, and, if the ENVIRONMENT directive has been set, it will also generate a Pascal environment file CatIO.pen.

Next, compile the main program:

```
$ pascal CatReac
```

This will generate object code, CatReac.obj.

Linking a Program

For execution, all of the object code (.obj files) must be linked together into an executable image. This is done with the link command:

```
$ link CatReac, CatIO, AidMod
```

This creates the image CatReac.exe, which can be executed as shown in the following sections.

Running a Program

A program can be run in two ways, either interactive or batch. For an interactive run, use the command procedure runjob as follows:

```
$ @runjob CatReac
```

The command procedure runjob is a command file runjob.com, which contains the following instructions:

```
$ set default [YOUR.DIRECTORY]
$ assign 'P1'.dat sys$input
$ assign 'P1'.out sys$output
$ run 'P1'
```

The command script accomplishes the following: First, the default directory is set to [YOUR.DIRECTORY]. Next, the the file CatReac.dat is defined as the standard input (since 'P1' in this case is CatReac), CatReac.out is defined as the standard output, and, finally, the program CatReac.exe is executed.

Alternatively, the program can be run in batch mode. For this, another command file is

created, runbatch.com:

```
$ submit/(parameters='P1') runjob
```

Thus, for a batch job, the command

```
$ @batchjob CatReac
```

will accomplish the same thing. For more details about writing and running Pascal programs on VAX machines, see the reference manuals for the current version.

Appendix B-1: Source Listing for Flow-Through Porous Electrode Program used in Part 1

On the following pages is listed the source code for the program used for the model calculations of Part 1. The main routine calls the subprogram BandShell, which is described in Part 2 of this dissertation and which is listed in the appendix that follows, appendix A-2. Following the listing of the program is the code for the input/output module, a sample data file, and a sample output file for a typical run. (Note: Although the program shown here is called FlowThru, it is different from the program called FlowThru shown in Part 2. Both programs, however, use the same model of the electrode, and the differences are outlined in section 2.10.2.)

Source Listing for Program FlowThru

```
[INHERIT('AidMod.pen','FlowIO.pen')]
{...

    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++

        Program Title : FlowThru
        Written By :    Michael Matlosz

        Date written : May 11, 1984
          Date updated : September 12, 1984
          Further updated : December 6, 1984

        Purpose:  This program calculates the concentration and potential
                  distribution in a flow-through porous electrode according
                  to the model of Trainham and Newman ( J. ELECTROCHEMICAL
                  SOCIETY, 124, p. 1528 (1977) ).

        Variable Number 1  == dimensionless concentration
        Variable Number 2  == dimensionless potential

        Axial diffusion and dispersion are included.

    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++

                                                                ...}


program FlowThru( input, output );


const   nEqns = 2;
        ImageFirstPoint = true;
        ImageLastPoint = true;
        FactorIncrement = 1e-6;
          AbsoluteIncrement = 1e-6;
        ReduceTimeOption = true;



var     Guess, FinalResult, Deviation, Residual : ValueArray;
          Profile : ValueArray;

        jMax, ItMax, j : integer;
        Tolerance, ThGuess, EtGuess : RealNumber;


        AlphaL, P1, P2, IStar, DP, C1, C2, C3 : RealNumber;
          P3, P4, P5, P6 : RealNumber;
        yMin, yMax : RealNumber;

          Configuration : ConfigChoice;

          IStarValues : ResultArray;
```

```
function Equation( i, j : integer;
                   y, h : RealNumber;
                   var NewResult : ValueArray;
                   function cInterp( k : integer;
                                     y : RealNumber;
                                     var Result : ValueArray ) : RealNumber;
                   function dnFdyn( n : integer;
                                    function F( y : RealNumber ) : RealNumber;
                                    y : RealNumber;
                                    Approx : DiffApprox ) : RealNumber
                                                                  ) : RealNumber;

        function u ( y : RealNumber ) : RealNumber;
        begin   u := cInterp(1,y,NewResult)          end;

        function v ( y : RealNumber ) : RealNumber;
        begin   v := cInterp(2,y,NewResult)          end;

        function dudy ( y : RealNumber ) : RealNumber;
        begin   dudy := dnFdyn(1,u,y,CenDiff)        end;

        function dvdy ( y : RealNumber ) : RealNumber;
        begin   dvdy := dnFdyn(1,v,y,CenDiff)        end;

        function d2udy2 ( y : RealNumber ) : RealNumber;
        begin   d2udy2 := dnFdyn(2,u,y,CenDiff)      end;

        function d2vdy2 ( y : RealNumber ) : RealNumber;
        begin   d2vdy2 := dnFdyn(2,v,y,CenDiff)      end;


        function T ( y : RealNumber ) : RealNumber;
        begin   T := exp( u(y) )             end;

        function E ( y : RealNumber ) : RealNumber;
        begin   E := v(y)          end;

        function dTdy ( y : RealNumber ) : RealNumber;
        begin    dTdy := exp( u(y) )*dudy(y)         end;

        function dEdy ( y : RealNumber ) : RealNumber;
        begin   dEdy := dvdy(y)              end;

        function d2Tdy2 ( y : RealNumber ) : RealNumber;
        begin   d2Tdy2 := exp( u(y) )*( sqr( dudy(y) ) + d2udy2(y) )  end;

        function d2Edy2 ( y : RealNumber ) : RealNumber;
        begin   d2Edy2 := d2vdy2(y)        end;
```

```
        function JR ( y : RealNumber ) : RealNumber;
        begin
                JR := ( T(y) - P1*exp( C1*E(y) ) )/( 1 + exp( E(y) ) )
        end;

        function JS ( y : RealNumber ) : RealNumber;
        begin
                JS := P3*exp( -C2*E(y) )*( 1 - P4*exp( C3*E(y) ) )
        end;

        function N ( y : RealNumber ) : RealNumber;
        begin
                N := ( 1/P2 )*dEdy(y) + T(y) - DP*dTdy(y)
        end;



function UpStrmBC( Eqn : integer ) : RealNumber;

var     EStope : RealNumber;

begin
        case Configuration of

        UD:     EStope :=  P5*IStar;
        DU:     EStope :=  P6*IStar;
        UU:     EStope := -P2*IStar;
        DD:     EStope :=   0

        end; { Configuration cases }


      case Eqn of

      1:      UpStrmBC := DP*dTdy(yMin) - T(yMin) + 1;  { = 0 }

       2:       UpStrmBC := N(h/2)
                                - ( EStope/P2 + T(yMin) - DP*dTdy(yMin) )
                                - (h/2)*JS(yMin)
                                                { = 0 }

      end  { Eqn cases }

end;  { UpStrmBC }


function DnStrmBC( Eqn : integer ) : RealNumber;

var     EStope : RealNumber;

begin
```

```
        case Configuration of

        UD:     ESlope := -P6*IStar;
        DU:     ESlope := -P5*IStar;
        UU:     ESlope :=  0;
        DD:     ESlope :=  P2*IStar

        end; { Configuration cases }

      case Eqn of

      1:    DnStrmBC := dTdy(yMax); { = 0 }

       2:       DnStrmBC := dEdy(yMax) - ESlope  { = 0 }

      end  { Eqn cases }

end; { DnStrmBC }


function DiffEQ( Eqn : integer; y : RealNumber ) : RealNumber;

begin

      case Eqn of

       1:       if ( DP <> 0 ) then
                        DiffEQ := DP*d2Tdy2(y) - JR(y) - dTdy(y)  { = 0 }
                else    DiffEQ := { nothing }  - JR(y) - dTdy(y); { = 0 }

       2:       DiffEQ := N(y + h/2) - N(y - h/2) - h*JS(y)  { = 0 }

       end  { Eqn cases }

end; { DiffEQ }


begin { body of Equation }

      if ( j = 1 ) then
                case i of

                1:      Equation := DiffEQ(1,yMin);
                2:      Equation := DiffEQ(2,yMin)

                end  { i cases }

       else if ( j = 2 ) then
                case i of

                1:      Equation := UpStrmBC(1);
                2:      Equation := UpStrmBC(2)

                end  { i cases }

       else if ( j > 2 ) and ( j < jMax - 1 ) then
```

```
                case i of

                1:      Equation := DiffEQ(1,y);
                2:      Equation := DiffEQ(2,y)

                end  { i cases }

        else if ( j = jMax - 1 ) then
                case i of

                1:      if (DP = 0) then
                                Equation := DiffEQ(1,yMax - h/2)
                        else    Equation := DiffEQ(1,yMax);

                2:      Equation := DiffEQ(2,yMax)

                end  { i cases }

        else if ( j = jMax ) then
                case i of

                1:      if (DP = 0) then
                                Equation := DiffEQ(1,yMax)
                        else    Equation := DnStrmBC(1);

                2:      Equation := DnStrmBC(2)

                end  { i cases }

        else    writeln('NODE OUT OF RANGE!')

end;  { Equation }



function Converged( function y( j : integer ) : RealNumber;
                    h : RealNumber;
                    var NewResult, Deviation : ValueArray;
                    var Residual : ValueArray;
                  function cInterp( k : integer;
                                    y : RealNumber;
                                    var Result : ValueArray ) : RealNumber;
                  function dnFdyn( n : integer;
                                   function F( y : RealNumber ) : RealNumber;
                                   y : RealNumber;
                                   Approx : DiffApprox ) : RealNumber
                                                                ) : boolean;

var     k, j : integer;
        absValue, absDeviation : RealNumber;

begin
        Converged := true;


        for k := 1 to nEqns do
            for j := 1 to jMax do
              begin
                absValue := abs(NewResult[k,j]);
```

```
                absDeviation := abs(Deviation[k,j]);

                if ( absValue <> 0 ) then
                        if ( abs(Deviation[k,j]) > Tolerance*absValue ) then
                                        Converged := false
            end

end;  { Converged }




procedure NonBandCalcs( LastIteration : boolean;
                        Iteration, CPUTime : integer;
                            function y( j : integer ) : RealNumber;
                            h : RealNumber;
                        var NewResult, Deviation : ValueArray;
                            var Residual : ValueArray;
                        function cInterp( k : integer;
                                          y : RealNumber;
                                          var Result : ValueArray ) : RealNumber;
                        function dnFdyn( n : integer;
                                         function F( y : RealNumber ) : RealNumber;
                                         y : RealNumber;
                                         Approx : DiffApprox ) : RealNumber );

var     j : integer;


function EtaPrime( y : RealNumber ) : RealNumber;
begin
        EtaPrime := cInterp(2,y,NewResult)
end;

begin


        if ( (Iteration = 0) and (Point = 1) ) or ( LastIteration ) then
        begin

                for j := 1 to jMax do
                begin
                        Profile[1,j] := exp( NewResult[1,j] );
                        Profile[2,j] := NewResult[2,j]
                end;


                if ( (Iteration = 0) and (Point = 1) ) then
                        writeln(' ':25,' -- Initial Guess -- ')
                else    writeln(' ':25,' -- Results (Point ',Point:1,') -- ');

                Printout( Profile, CPUTime, Iteration, jMax,
                                   Configuration, EtaPrime, AlphaL, IStarValues);

        end
end;
```

```
begin   { body of FlowThru }

        ReadSetWriteParameters( jMax, ItMax, Tolerance, ThGuess, EtGuess,
                P1, P2, P3, P4, P5, P6, DP, AlphaL, C1, C2, C3,
                Configuration, IStarValues );

        yMin := 0;          yMax := AlphaL;

        for j := 1 to jMax do
        begin
                Guess[1,j] := ln(ThGuess);
                Guess[2,j] := EtGuess
        end;



        for Point := 1 to NumberOfPoints do
        begin

            IStar := IStarValues[Point];

          BandShell( nEqns, jMax, ItMax,
                yMin, yMax, FactorIncrement, AbsoluteIncrement,
                ImageFirstPoint, ImageLastPoint, ReduceTimeOption,
                Guess, FinalResult, Deviation, Residual,
                Equation, Converged, NonBandCalcs );

            Guess := FinalResult

        end;

        PrintSummary(IStarValues)

end   { FlowThru }
```

Source Listing for FlowIO

```
[INHERIT('AidMod.pen'),ENVIRONMENT('FlowIO.pen')]
module FlowIO(input, output);

const   F = 96487; { coul/eq }
        R = 8.314; { J/mol-K }

type    ResultArray = array [1..40] of RealNumber;
        ConfigChoice = ( UU, UD, DU, DD );

var     a, Alpha, AlphaAR, AlphaAS, AlphaCR, AlphaCS,
                cClf, cClref, cHf, cHref, cRf, cRref,
                D0, Da, DeltaU, DR,
                epsilon,
                gammaR, gammaRCl, gammaSH, gammaSH2,
                i, ioRf, ioRref, ioSf, ioSref,
                kappa, kappa0, kmR,
                L, n,
                pH, PH2, PH2ref, pHref,
                RFixed, rhoZero,
                sigma, sR,
                T, tau,
                UR, Ure, URtheta, US, UStheta,
                v                                   : RealNumber;


        Point, NumberOfPoints : integer;

        iValues, EtaExit, OhmicDrop, AreaSpecificResistance : ResultArray;

        DateString, TimeString : packed array [1..11] of char;

        ConfigString, Comment : String;



procedure SetConfig( var Configuration : ConfigChoice;
                     ConfigString : String  );

var     ch1, ch2 : char;
        i : integer;

begin

        Configuration := UD;  {... UD is the default }

                {... ignore leading blanks, find first two characters }

        i := 0;

        repeat  i := i + 1  until (ConfigString.chars[i] <> ' ');

        ch1 := ConfigString.chars[i];
        ch2 := ConfigString.chars[i+1];
```

```
                {... set Config according to input string }

        case ch1 of

        'u','U':  if (ch2 = 'd') or (ch2 = 'D') then
                        Configuration := UD

                  else if (ch2 = 'u') or (ch2 = 'U') then
                        Configuration := UU;

        'd','D':  if (ch2 = 'u') or (ch2 = 'U') then
                        Configuration := DU

                  else if (ch2 = 'd') or (ch2 = 'D') then
                        Configuration := DD

        end  { ch1 cases }
end;  { SetConfiguration }


procedure WriteSelection( Configuration : ConfigChoice );

begin

        case Configuration of

        UU:     begin
                        write(' UU ');
                        LF(1); TB(20); write('Counterelectrode = Upstream');
                        LF(1); TB(20); write('Current Collector = Upstream')
                end;

        UD:     begin
                        write(' UD ');
                        LF(1); TB(20); write('Counterelectrode = Upstream');
                        LF(1); TB(20); write('Current Collector = Downstream')
                end;

        DU:     begin
                        write(' DU ');
                        LF(1); TB(20); write('Counterelectrode = Downstream');
                        LF(1); TB(20); write('Current Collector = Upstream')
                end;

        DD:     begin
                        write(' DD ');
                        LF(1); TB(20); write('Counterelectrode = Downstream');
                        LF(1); TB(20); write('Current Collector = Downstream')
                end

        end  { Configuration cases }

end;  { WriteSelection }



procedure ReadSetWriteParameters( var jMax, ItMax : integer;
```

```
                    var Tolerance, ThGuess, EtGuess : RealNumber;
                    var P1, P2, P3, P4, P5, P6,
                          DP, AlphaL, C1, C2, C3 : RealNumber;
                    var Configuration : ConfigChoice;
                    var iStarValues : ResultArray          );

var     Beta : RealNumber;
        P2Term, P3Term, P4Term : array [1..3] of RealNumber;

        h, Criterion : RealNumber;
        DispersionIncluded : boolean;

begin

        {... Step 1.  Read the data ...}


        RS(Comment);

        RI(jMax);   RI(ItMax);
        RR(Tolerance);

        RR(ThGuess);  RR(EtGuess);

        RR(n);
        RR(L);
        RR(T);
        RR(D0);
        RR(epsilon);
        RR(a);
        RR(tau);
        RR(v);
        RR(kappa0);
        RR(sigma);
        RR(RFixed);
        RR(sR);

        RR(cRf); { mol/cc }
        RR(cClf); { mol/cc }
        RR(pH);
        RR(PH2);  { atm }

        RR(kmR);
        RR(AlphaAR);        RR(AlphaCR);
        RR(AlphaAS);        RR(AlphaCS);
        RR(ioRref);         RR(cRref);   RR(cClref);
        RR(ioSref);         RR(pHref);   RR(PH2ref);

        RR(URtheta);
        RR(UStheta);
        RR(Ure);
        RR(rhoZero);


        RS(ConfigString);


        RI(NumberOfPoints);

        for Point := 1 to NumberOfPoints do RR(iStarValues[Point]);
```

```
{... Step 2. Set the parameters ...}


cHf := 10**( -pH - 3 );  {... in mol/cc ...}
cHref := 10**( -pHref - 3 );  {... in mol/cc ...}


UR := URtheta - Ure + (-sR*R*T/(n*F))*ln(cRf/rhoZero)
                                - (4*R*T/(n*F))*ln(cClf/rhoZero);

US := UStheta - Ure + (R*T/F)*ln(cHf/rhoZero)
                                - (R*T/(2*F))*ln(PH2);

DeltaU := US - UR;


Da := 3*v*(1-epsilon)/(a*epsilon);

DR := D0/sqr(tau);
DP := epsilon*( DR + Da )*a*kmR/sqr(v);

kappa := kappa0*( epsilon**1.5 );


SetConfig( Configuration, ConfigString );


gammaR := -sR + ( AlphaCR/n )*sR;
gammaRCl := ( AlphaCR/n )*4;

  gammaSH := 1 - AlphaCS;
  gammaSH2 := AlphaCS/2;

ioRf := ioRref*( (cRf/cRref)**(gammaR) )*( (cClf/cClref)**(gammaRCl) );
ioSf := ioSref*( (cHf/cHref)**(gammaSH) )*( (PH2/PH2ref)**(gammaSH2) );


Beta := AlphaAR/AlphaCR;

C1 := 1 + Beta;
C2 := AlphaCS/AlphaCR;
C3 := ( AlphaAS + AlphaCS )/AlphaCR;


P1 := ( -sR*ioRf/(n*F*kmR*CRf) )**(C1);


P2Term[1] := ( AlphaCR*n*sqr(F*v)*CRf )/( sR*a*kmR*R*T );
P2Term[2] := (1/kappa) + (1/sigma);

P2 := P2Term[1]*P2Term[2];
```

```
P3Term[1] := ( -sR*ioSf )/( n*F*kmR*CRf );
P3Term[2] := exp( AlphaCS*F*DeltaU/(R*T) );
P3Term[3] := ( ( -n*F*kmR*CRf )/( sR*ioRf ) )**(C2);

P3 := P3Term[1]*P3Term[2]*P3Term[3];


P4Term[1] := ( ( -sR*ioRf )/( n*F*kmR*CRf ) )**(C3);
P4Term[2] := exp( -F*( AlphaAS + AlphaCS )*DeltaU/(R*T) );

P4 := P4Term[1]*P4Term[2];


P5     := ( -sigma*P2 )/( sigma + kappa );


P6     := P5*( kappa/sigma );


for Point := 1 to NumberOfPoints do
        iValues[Point] := ( (n*F*v*CRf)/sR )*iStarValues[Point];


Alpha := a*kmR/v;
AlphaL := ( a*kmR*L )/v;

h := AlphaL/(jMax - 3);
Criterion := 1 - h/(2*DP);

if ( Criterion <= 0 ) then
        DispersionIncluded := false
else    DispersionIncluded := true;


{... Step 3.  Print the data and parameters ...}


LF(5); TB(25); write('Flow-Thru Porous Electrode Program');
LF(2); TB(25); write('            written  by');
LF(2); TB(25); write('           Michael Matlosz');
LF(2); TB(25); write('(Date Written : 14 May 1984 ');
LF(1); TB(25); write('      Revised : 12 September 1984)');
LF(1); TB(25); write('Revised Again : 6 December 1984)');

Date(DateString);
Time(TimeString);

LF(2); TB(45); write('Program Begun at ',TimeString);
LF(1); TB(45); write('                on ',DateString);


LF(5); TB(10); write('Comment --- '); WS(Comment);


LF(5); TB(15); write('Input Parameters -- ');
```

```
LF(2); TB(17); write('Number Of Mesh Points = '); WI(jMax,1);
LF(1); TB(17); write('Convergence Tolerance = '); WR(Tolerance,10,5);
LF(1); TB(17); write('Iteration Limit = '); WI(ItMax,1);

LF(3); TB(17); write('Electrode Length (L) = ');
                 WR(L,8,5); write(' cm');

LF(1); TB(17); write('Temperature (T) = ');
                 WR(T,9,5);  write(' K');

LF(2); TB(17); write('Void Fraction (epsilon) = ');  WR(epsilon,7,5);
LF(1); TB(17); write('Tortuosity (tau) = '); WR(tau,7,5);

LF(1); TB(17); write('Interfacial Area per Volume (a) = ');
                                  WR(a,10,5); write(' cm2/cm3');

LF(1); TB(17); write('Fluid Velocity (v) = ');
                                  WR(v,10,5); write(' cm/s');

LF(2); TB(17); write('Solvent Density (rhoZero) = ');
                                  WR(rhoZero,10,5); write(' kg/cm3');

LF(2); TB(17); write('Free-Solution Reactant Diffusivity (D0) = ');
                                  WR(D0,10,5); write(' cm2/s');

LF(1); TB(17); write('Effective Reactant Diffusivity (DR) = ');
                                  WR(DR,10,5); write(' cm2/s');

LF(1); TB(17); write('Axial Dispersion Coefficient (Da) = ');
                                  WR(Da,10,5); write(' cm2/s');

LF(2); TB(17); write('Free Solution Conductivity (kappa0) = ');
                                  WR(kappa0,10,5); write(' mho/cm');

LF(2); TB(17);
        write('Effective Solution Conductivity (kappa) = ');
                                  WR(kappa,10,5); write(' mho/cm');

LF(3); TB(17); write('Electrode conductivity (sigma) = ');
                                  WR(sigma,10,5); write(' mho/cm');
LF(2); TB(17); write('Fixed Resistance (RFixed) = ');
                                  WR(RFixed,10,5); write(' ohm-cm2');

LF(3); TB(17);
        write('Reactant stoichiometric coefficient (sR) = ');
                                  WR(sR,8,4);

LF(3); TB(17); write('Reactant Feed Concentration (cRf) = ');
                                  WR(cRf,10,5); write(' mol/cm3');

LF(1); TB(17); write('Chloride-Ion Concentration (cClf) = ');
                                  WR(cClf,10,5); write(' mol/cm3');

LF(1); TB(17); write('pH of Feed Solution = ');
                                  WR(pH,10,5);

LF(1); TB(17); write('Hydrogen Partial Pressure (PH2) = ');
                                  WR(PH2,10,5); write(' atm');


LF(3); TB(17); write('Reactant Mass-Transfer Coefficient (kmR) = ');
                                  WR(kmR,10,5); write(' cm/s');
```

```
LF(2); TB(17); write('alpha (a*kmR/v) = ');
                        WR(Alpha,10,5); write(' 1/cm');
LF(1); TB(17); write('v/(a*kmR)  (penetration depth) = ');
                        WR(( 1/Alpha ),10,5); write(' cm');

LF(3); TB(17); write('Transfer Coefficients:');
LF(2); TB(17); write('Main reaction');
LF(1); TB(25); write('AlphaAR = '); WR(alphaAR,10,5);
LF(1); TB(25); write('AlphaCR = '); WR(alphaCR,10,5);
LF(1); TB(17); write('Side reaction');
LF(1); TB(25); write('AlphaAS = '); WR(alphaAS,10,5);
LF(1); TB(25); write('AlphaCS = '); WR(alphaCS,10,5);

LF(2); TB(17); write('gammaR = ');  WR(gammaR,10,5);
LF(1); TB(17); write('gammaRCl = '); WR(gammaRCl,10,5);
LF(1); TB(17); write('gammaSH = '); WR(gammaSH,10,5);
LF(1); TB(17); write('gammaSH2 = '); WR(gammaSH2,10,5);


LF(3); TB(17); write('Exchange-Current Densities:');

LF(3); TB(25); write('Main Reaction (ioRref) = ');
                        WR(ioRref,10,5); write(' A/cm2');
LF(1); TB(30); write(' at  cRref = ');
                        WR(cRref,10,5); write(' mol/cm3');
LF(1); TB(30); write('     cClref = ');
                        WR(cClref,10,5); write(' mol/cm3');

LF(2); TB(25); write('Main Reaction (ioRf) = ');
                        WR(ioRf,10,5); write(' A/cm2');
LF(1); TB(30); write(' at  cRf = '); WR(cRf,10,5); write(' mol/cm3');
LF(1); TB(30); write('     cClf = '); WR(cClf,10,5); write(' mol/cm3');


LF(3); TB(25); write('Side Reaction (ioSref) = ');
                        WR(ioSref,10,5); write(' A/cm2');
LF(1); TB(30); write(' at  pHref = '); WR(pHref,10,5);
LF(1); TB(30); write('     PH2ref = '); WR(PH2ref,10,5); write(' atm');

LF(2); TB(25); write('Side Reaction (ioSf) = ');
                        WR(ioSf,10,5); write(' A/cm2');
LF(1); TB(30); write(' at  pH = '); WR(pH,10,5);
LF(1); TB(30); write('     PH2 = '); WR(PH2,10,5); write(' atm');

LF(4); TB(17); write('URtheta = ');
                        WR(URtheta,10,5); write(' V');
LF(1); TB(17); write('UStheta = ');
                        WR(UStheta,10,5); write(' V');
LF(1); TB(17); write('Ure = ');
                        WR(Ure,10,5); write(' V');

LF(1); TB(17); write('UR (at cRf and cClf) = ');
                        WR(UR,10,5); write(' V');
LF(1); TB(17); write('US (at pH and PH2) = ');
                        WR(US,10,5); write(' V');
LF(2); TB(17); write('DeltaU (US - UR) = ');
                        WR(DeltaU,10,5); write(' V');

LF(3); TB(17); write('Electrode Configuration:');
                  WriteSelection(Configuration);


LF(3);
```

```
        TB(25); write('Point');
        TB(8); write('iStar');
        TB(14); write('i (A/cm2)');

        LF(2);

        for Point := 1 to NumberOfPoints do
        begin
                TB(26); WI(Point,2);
                TB(6); WR(iStarValues[Point],10,5);
                TB(6); WR(iValues[Point],10,5);
                LF(1)

        end;


        LF(3); TB(20); write('Reduced Parameters --- ');
        LF(2); TB(30); write('P1 = '); WR(P1,10,5);
        LF(1); TB(30); write('P2 = '); WR(P2,10,5);
        LF(1); TB(30); write('P3 = '); WR(P3,10,5);
        LF(1); TB(30); write('P4 = '); WR(P4,10,5);
        LF(1); TB(30); write('P5 = '); WR(P5,10,5);
        LF(1); TB(30); write('P6 = '); WR(P6,10,5);

        LF(2); TB(30); write('C1 ( 1 + AlphaAR/AlphaCR ) = '); WR(C1,10,5);
        LF(1); TB(30); write('C2 ( AlphaCS/AlphaCR ) = '); WR(C2,10,5);
        LF(1); TB(30); write('C3 ( (AlphaAS + AlphaCS)/AlphaCR ) = ');
                                                        WR(C3,10,5);

        LF(2); TB(30); write('AlphaL = '); WR(AlphaL,10,5);
        LF(1); TB(30); write('DP = '); WR(DP,10,5);

        LF(3); TB(20); write('h (mesh size) = '); WR(h,10,5);
        LF(2); TB(20); write('1 - h/(2*DP) = ');  WR(Criterion,10,5);
        LF(1); TB(30);

                if DispersionIncluded then
                        write('--- Axial Dispersion Included')
                else
                begin
                        write('--- Axial Dispersion Neglected');
                           LF(1); TB(30);
                        write('        ( DP = 0, km interpreted as kmBAR )');
                        DP := 0
                end;


        LF(15)

end;  { ReadSetWriteParameters }




procedure Printout( Result : ValueArray;
                CPUTime, Iterations, jMax : integer;
                Configuration : ConfigChoice;
                function EtaPrime( x : RealNumber ) : RealNumber;
```

```
                AlphaL : RealNumber;
                iStarValues : ResultArray );

const   ImageFirstPoint = true;
        ImageLastPoint = true;


function Eta( x : RealNumber ) : RealNumber;

var     y : RealNumber;

begin
        y := Alpha*x;

        Eta := ( R*T/(AlphaCR*F) )*( EtaPrime(y)
                                - ln( -n*F*kmR*cRf/(sR*ioRf) ) ) + UR

end;


function Phi1( x : RealNumber ) : RealNumber;

var     Distribution : RealNumber;
        i : RealNumber;

begin

        i := iValues[Point];

        case Configuration of

        UD:     Distribution := Eta(x) - Eta(L) + (i*L/kappa)*( x/L - 1);
        DU:     Distribution := Eta(x) - Eta(0) - (i/kappa)*x;
        UU:     Distribution := Eta(x) - Eta(0);
        DD:     Distribution := Eta(x) - Eta(L)

        end;  { Configuration cases }

        Phi1 := ( (kappa + sigma)/kappa )*Distribution

end;


function Phi2( x : RealNumber ) : RealNumber;

begin
        Phi2 := Phi1(x) - Eta(x)
end;


procedure PrintProfiles( Result : ValueArray; CPUTime,
                Iterations, jMax : integer;
                AlphaL : RealNumber );

var     j : integer;
        Value1, Value2 : RealNumber;

function z( y : RealNumber ) : RealNumber;
begin   z := y          end;
```

```
begin

        LF(2); TB(5); write('Iterations = '); WI(Iterations,1);
        LF(1); TB(5); write('CPU Time = ');
                        WI(CPUTime,1); write(' milliseconds');
        LF(3); TB(25); write('Profile Listing'); LF(2);
        TB(5); write('j');
        TB(9); write('y');
        TB(15); write('Theta');
        TB(12); write('Eta-Prime');


        LF(2);


        ListPrint( Result, z, 2, jMax,
                        ImageFirstPoint, ImageLastPoint, 0, AlphaL );

        LF(5)

end; { PrintProfiles }



begin { Printout }


        LF(2); TB(42); write('iStar =  '); WR(iStarValues[Point],10,5);

        LF(1); TB(46); write('i =  '); WR(iValues[Point],10,5);
                                        write(' A/cm2');

        EtaExit[Point] := Eta(L);

        OhmicDrop[Point] := Eta(L) - Eta(0);


        if ( iValues[Point] <> 0 ) then
                AreaSpecificResistance[Point] :=
                        abs( OhmicDrop[Point]/iValues[Point] ) + RFixed

        else    AreaSpecificResistance[Point] := RFixed;


        LF(1); TB(40); write('EtaExit = ');
                WR(EtaExit[Point]*1000,8,4); write(' mV');

        LF(1); TB(22); write('Solution-Phase Ohmic Drop = ');
                        WR(OhmicDrop[Point]*1000,8,4); write(' mV');

        LF(1); TB(22); write(' Area-Specific Resistance = ');
                if ( iValues[Point] = 0 ) then
                        write(' Undefined')
                else
                begin
                        WR(AreaSpecificResistance[Point],8,4);
                        write(' ohm-cm2')
                end;

        LF(1);
```

```
        PrintProfiles( Result, CPUTime, Iterations, jMax, AlphaL )

end;  { Printout }


procedure PrintSummary ( iStarValues : ResultArray );

var     Point : integer;
        TimeString, DateString : packed array[1..11]of char;


begin
        LF(5);

        TB(25); write('Summary of Results');
        LF(3);

        TB(5); write('Point');
        TB(5); write('I (A/cm2)');
        TB(10); write('EtaExit (mV)');
        TB(4); write('Solution-Phase Ohmic Drop (mV)');

        LF(2);

        for Point := 1 to NumberOfPoints do
        begin
                TB(6); WI(Point,2);
                TB(4); WR(iValues[Point],10,5);
                TB(7); WR(EtaExit[Point]*1000,10,4);
                TB(9); WR(OhmicDrop[Point]*1000,10,4);
                LF(1)
        end;

        LF(2);

        TB(5); write('Point');
        TB(5); write('EtaExit (mV)');
        TB(7); write('iStar');
        TB(5); write('Area-Specific Resistance (ohm-cm2)');

        LF(2);

        for Point := 1 to NumberOfPoints do
        begin
                TB(6); WI(Point,2);
                TB(6); WR(EtaExit[Point]*1000,10,4);
                TB(2); WR(iStarValues[Point],10,5);
                TB(10);
                    if ( iValues[Point] = 0 ) then
                        write('Undefined')
                    else WR(AreaSpecificResistance[Point],10,4);
                LF(1)

        end;

        Date(DateString);  Time(TimeString);

        LF(5); TB(25); write('FlowThru Program Stopped at ',TimeString);
        LF(1); TB(25); write('                              on ',DateString);
```

```
        LF(5); TB(25); write('FlowThru End');
        LF(2)
end;  { PrintSummary }
end.  { FlowIO module }
```

Sample Data File for FlowThru

++++++++++++ Data file for flow-through porous
electrode program ++++++++++++++++++

Comment = < Case 1. Hg removal. >

jMax = 41
ItMax = 10

Tolerance = 1e-8

Theta-Guess = 1
Eta-Guess = +4

n = 2
L = 12.7 cm
T = 298.15 K

Do = 1.0e-5 cm2/s
epsilon = 0.97
a = 66 cm2/cm3
tau = 1.0

v = 0.0255 cm/s

kappa-zero = 0.199 mho/cm
sigma = 1.73 mho/cm

Rfixed = 30 ohm-cm2

sR = -1
cRf = 2.273e-7 mol/cm3
cClf = 4.3e-3 mol/cm3
pH = 4
PH2 = 5e-7 atm

kmR = 2.102e-4 cm/s

alphaAR = 1.4
alphaCR = 0.6

alphaAS = 0.5
alphaCS = 0.5

ioRref = 3e-7 A/cm2
cRref = 5e-7 mol/cm3
cClref = 3.8e-3 mol/cm3

ioSref = 2e-6 A/cm2
pHref = 0
PH2ref = 1 atm

URtheta = 0.4138 V (mercuric-chloride complex reduction)

UStheta = 0.000 V (hydrogen evolution)
Ure = 0.2415 V (calomel (sat'd KCl))

rhoZero = 1.14e-3 kg/cm3

Electrode Configuration = <UD>

Number of Points = 2

IStar
= 0.0
= 0.5

++++++++++++++ End of Data File ++++++++++++++++++++

Sample Output from FlowThru

Flow-Thru Porous Electrode Program

written by

Michael Matlosz

(Date Written : 14 May 1984
Revised : 12 September 1984)
Revised Again : 6 December 1984)

Program Begun at 23:22:31.52
on 11-MAR-1985

Comment --- Case 1. Hg removal.

Input Parameters -

Number Of Mesh Points = 41
Convergence Tolerance = 1.0000E-08
Iteration Limit = 10

Electrode Length (L) = 12.700 cm
Temperature (T) = 298.15 K

Void Fraction (epsilon) = 0.97000
Tortuosity (tau) = 1.0000
Interfacial Area per Volume (a) = 66.000 cm2/cm3
Fluid Velocity (v) = 0.025500 cm/s

Solvent Density (rhoZero) = 0.0011400 kg/cm3

Free-Solution Reactant Diffusivity (D0) = 1.0000E-05 cm2/s
Effective Reactant Diffusivity (DR) = 1.0000E-05 cm2/s
Axial Dispersion Coefficient (Da) = 3.5848E-05 cm2/s

Free Solution Conductivity (kappa0) = 0.19900 mho/cm

Effective Solution Conductivity (kappa) = 0.19011 mho/cm

Electrode conductivity (sigma) = 1.7300 mho/cm

Fixed Resistance (RFixed) = 30.000 ohm-cm2

Reactant stoichiometric coefficient (sR) = -1.000

Reactant Feed Concentration (cRf) = 2.2730E-07 mol/cm3
Chloride-Ion Concentration (cClf) = 0.0043000 mol/cm3
pH of Feed Solution = 4.0000
Hydrogen Partial Pressure (PH2) = 5.0000E-07 atm

Reactant Mass-Transfer Coefficient (kmR) = 2.1020E-04 cm/s

alpha (a*kmR/v) = 0.54405 1/cm
v/(a*kmR) (penetration depth) = 1.8381 cm

Transfer Coefficients:

Main reaction
AlphaAR = 1.4000
AlphaCR = 0.60000
Side reaction
AlphaAS = 0.50000
AlphaCS = 0.50000

gammaR = 0.70000
gammaRCl = 1.2000
gammaSH = 0.50000
gammaSH2 = 0.25000

Exchange-Current Densities:

Main Reaction (ioRref) = 3.0000E-07 A/cm2
at cRref = 5.0000E-07 mol/cm3
cClref = 0.0038000 mol/cm3

Main Reaction (ioRf) = 2.0039E-07 A/cm2
at cRf = 2.2730E-07 mol/cm3
cClf = 0.0043000 mol/cm3

Side Reaction (ioSref) = 2.0000E-06 A/cm2
at pHref = 0.00000
PH2ref = 1.0000 atm

Side Reaction (ioSf) = 5.3183E-10 A/cm2
at pH = 4.0000
PH2 = 5.0000E-07 atm

URtheta = 0.41380 V
UStheta = 0.00000 V
Ure = 0.24150 V
UR (at cRf and cClf) = -0.0053591 V
US (at pH and PH2) = -0.29512 V

DeltaU (US - UR) = -0.28976 V

Electrode Configuration: UD
Counterelectrode = Upstream
Current Collector = Downstream

Point	iStar	i (A/cm2)
1	0.00000	0.00000
2	0.50000	-5.5925E-04

Reduced Parameters --

P1 = 2.8653E-06
P2 = -0.28032
P3 = 4.9843E-06
P4 = 133.93
P5 = 0.25256
P6 = 0.027754

C1 (1 + AlphaAR/AlphaCR) = 3.3333
C2 (AlphaCS/AlphaCR) = 0.83333
C3 ((AlphaAS + AlphaCS)/AlphaCR) = 1.6667

AlphaL = 6.9094
DP = 9.4883E-04

h (mesh size) = 0.18183

1 - h/(2*DP) = -94.816
--- Axial Dispersion Neglected
(DP = 0, km interpreted as kmBAR)

-- Initial Guess --

iStar = 0.00000
i = 0.00000 A/cm2
EtaExit = 1.969 mV
Solution-Phase Ohmic Drop = 0.0000 mV
Area-Specific Resistance = Undefined

Iterations = 0
CPU Time = 1050 milliseconds

Profile Listing

j	y	Theta	Eta-Prime
1	-0.18183	1.000000	4.000000
2	0.00000	1.000000	4.000000
3	0.18183	1.000000	4.000000
4	0.36365	1.000000	4.000000
5	0.54548	1.000000	4.000000
6	0.72731	1.000000	4.000000
7	0.90913	1.000000	4.000000
8	1.0910	1.000000	4.000000
9	1.2728	1.000000	4.000000
10	1.4546	1.000000	4.000000
11	1.6364	1.000000	4.000000
12	1.8183	1.000000	4.000000
13	2.0001	1.000000	4.000000
14	2.1819	1.000000	4.000000
15	2.3637	1.000000	4.000000
16	2.5456	1.000000	4.000000
17	2.7274	1.000000	4.000000
18	2.9092	1.000000	4.000000
19	3.0910	1.000000	4.000000
20	3.2729	1.000000	4.000000
21	3.4547	1.000000	4.000000
22	3.6365	1.000000	4.000000
23	3.8184	1.000000	4.000000
24	4.0002	1.000000	4.000000
25	4.1820	1.000000	4.000000
26	4.3638	1.000000	4.000000
27	4.5457	1.000000	4.000000
28	4.7275	1.000000	4.000000
29	4.9093	1.000000	4.000000
30	5.0911	1.000000	4.000000
31	5.2730	1.000000	4.000000
32	5.4548	1.000000	4.000000
33	5.6366	1.000000	4.000000

34	5.8184	1.000000	4.000000
35	6.0003	1.000000	4.000000
36	6.1821	1.000000	4.000000
37	6.3639	1.000000	4.000000
38	6.5457	1.000000	4.000000
39	6.7276	1.000000	4.000000
40	6.9094	1.000000	4.000000
41	7.0912	1.000000	4.000000

– Results (Point 1) –

iStar = 0.00000
i = 0.00000 A/cm2
EtaExit = -15.53 mV
Solution-Phase Ohmic Drop = -0.1084 mV
Area-Specific Resistance = Undefined

Iterations = 5
CPU Time = 19340 milliseconds

Profile Listing

j	y	Theta	Eta-Prime
1	-0.18183	1.002648	3.593952
2	0.00000	1.000000	3.593958
3	0.18183	0.9973652	3.593952
4	0.36365	0.9947431	3.593936
5	0.54548	0.9921334	3.593911
6	0.72731	0.9895362	3.593877
7	0.90913	0.9869512	3.593834
8	1.0910	0.9843783	3.593783
9	1.2728	0.9818172	3.593725
10	1.4546	0.9792681	3.593660
11	1.6364	0.9767306	3.593590
12	1.8183	0.9742048	3.593514
13	2.0001	0.9716904	3.593433
14	2.1819	0.9691874	3.593347
15	2.3637	0.9666957	3.593259
16	2.5456	0.9642152	3.593166
17	2.7274	0.9617459	3.593072
18	2.9092	0.9592876	3.592975
19	3.0910	0.9568403	3.592877
20	3.2729	0.9544039	3.592778

21	3.4547	0.9519784	3.592678
22	3.6365	0.9495637	3.592579
23	3.8184	0.9471598	3.592480
24	4.0002	0.9447666	3.592383
25	4.1820	0.9423841	3.592287
26	4.3638	0.9400123	3.592194
27	4.5457	0.9376511	3.592103
28	4.7275	0.9353006	3.592016
29	4.9093	0.9329606	3.591933
30	5.0911	0.9306312	3.591854
31	5.2730	0.9283124	3.591780
32	5.4548	0.9260043	3.591711
33	5.6366	0.9237067	3.591649
34	5.8184	0.9214197	3.591593
35	6.0003	0.9191433	3.591544
36	6.1821	0.9168776	3.591503
37	6.3639	0.9146226	3.591470
38	6.5457	0.9123782	3.591446
39	6.7276	0.9101446	3.591431
40	6.9094	0.9079218	3.591426
41	7.0912	0.9057097	3.591431

-- Results (Point 2) --

iStar = 0.50000
i = -5.5925E-04 A/cm2
EtaExit = -76.02 mV
Solution-Phase Ohmic Drop = 12.89 mV
Area-Specific Resistance = 53.05 ohm-cm2

Iterations = 6
CPU Time = 22220 milliseconds

Profile Listing

j	y	Theta	Eta-Prime
1	-0.18183	1.024632	1.853839
2	0.00000	1.000000	1.877409
3	0.18183	0.9764356	1.899781
4	0.36365	0.9538535	1.921007
5	0.54548	0.9321900	1.941136
6	0.72731	0.9113745	1.960215
7	0.90913	0.8913553	1.978284

8	1.0910	0.8720731	1.995384
9	1.2728	0.8534857	2.011552
10	1.4546	0.8355428	2.026822
11	1.6364	0.8182098	2.041227
12	1.8183	0.8014436	2.054796
13	2.0001	0.7852158	2.067559
14	2.1819	0.7694891	2.079543
15	2.3637	0.7542399	2.090772
16	2.5456	0.7394360	2.101271
17	2.7274	0.7250576	2.111062
18	2.9092	0.7110767	2.120166
19	3.0910	0.6974767	2.128603
20	3.2729	0.6842329	2.136393
21	3.4547	0.6713315	2.143553
22	3.6365	0.6587505	2.150101
23	3.8184	0.6464785	2.156053
24	4.0002	0.6344959	2.161424
25	4.1820	0.6227931	2.166228
26	4.3638	0.6113525	2.170481
27	4.5457	0.6001662	2.174195
28	4.7275	0.5892183	2.177382
29	4.9093	0.5785022	2.180056
30	5.0911	0.5680035	2.182227
31	5.2730	0.5577169	2.183906
32	5.4548	0.5476292	2.185104
33	5.6366	0.5377360	2.185832
34	5.8184	0.5280254	2.186098
35	6.0003	0.5184937	2.185912
36	6.1821	0.5091301	2.185282
37	6.3639	0.4999316	2.184218
38	6.5457	0.4908884	2.182727
39	6.7276	0.4819978	2.180817
40	6.9094	0.4732510	2.178496
41	7.0912	0.4646460	2.175771

Summary of Results

Point	I (A/cm2)	EtaExit (mV)	Solution-Phase Ohmic Drop (mV)
1	0.00000	-15.53	-0.1084
2	-5.5925E-04	-76.02	12.89

Point	EtaExit (mV)	iStar	Area-Specific Resistance (ohm-cm2)
1	-15.53	0.00000	Undefined
2	-76.02	0.50000	53.05

FlowThru Program Stopped at 23:24:59.39
on 11-MAR-1985

FlowThru End

Appendix B-2: Source Listings of BandAid, BandShell and Input/Output Routines for the Examples of Part 2

In the pages that follow, all of the computer code necessary to run the examples of Part 2 is presented. The listing of each separate subprogram or group of subprograms begins on a new page with a heading containing the title and a suggested file name for the text.

Source Code for Procedure BandShell (File: BandShell.pas)

{...

++

Procedure Title: BandShell -- Version 3, Release 3.2

Written By: Michael Matlosz
Date: October 20, 1984

Procedure Summary: BandShell arranges information supplied by a main program so that the BAND algorithm of Newman can be used to solve a set of coupled, non-linear ordinary differential equations. (Newman's algorithm is contained in the subroutine BandCore, which is called by BandShell.) The solution to the problem (as determined by BandCore and returned to BandShell) is sent back to the main (user's) program.

Input: The procedure is intended to be used in conjunction with a main driver program that specifies information about the differential equations to be solved and the method of solution. A summary of the necessary information for the preparation of this driver program is listed below.

(Note: in some cases the GLOBAL definitions required for the driver routine may contain non-standard language features (most notably the function clock, and the type of the real numbers (e. g. double)), which have been chosen below for compatibility with VAX-11 Pascal, Version 2. If the routine is implemented using a different Pascal compiler, there may be some problems. If there are, consult a local system expert. The source listing produced here (i. e., the BandShell routine itself), however, is completely within the ISO (International Standards Organization) Standard Pascal Language Specification (i. e., the (Draft) ANSI/IEEE Pascal Standard, X3J9/81-093, adopted by the American National Standards Institute), and should require no modification for portability.):

GLOBAL CONST DECLARATIONS (must be declared in the driver program):

```
const   nLimit = (integer); -- limits the number of
                                    equations and unknowns.

        WidthLimit = (integer); -- maximum columns of the
                          augmented matrices of BandCore
                          ( = 2*nLimit + 1 ).
```

```
jLimit = (integer); -- limits the number of
                                    mesh points.

RLimit = (integer); -- limits the maximum number of
                                    iterations allowed.

SystemZero = (RealNumber); -- sets the smallest (absolute)
                              value allowed for a RealNumber.

SigDigits = (integer); -- indicates the maximum number
                          of significant digits for a
                          RealNumber.  (This is used to
                          determine the uncertainty in
                          the truncation used for evaluating
                          the mesh points in the interpolation
                          functions of BandShell.)
```

GLOBAL TYPE DECLARATIONS (must be declared in the driver program):

```
type    RealNumber = real;  ( OR  RealNumber = double;
                              OR  RealNumber = quad; )

        -- sets the type of the real numbers:
             real (7-digit accuracy)
             double (15-digit accuracy)
             quad (33-digit accuracy)

        ValueArray = array [1..nLimit,1..jLimit] of RealNumber;

        -- defines a type for value storage.

        DiffApprox = ( CenDiff, BackDiff, ForDiff,
                       Back3PtDiff, For3PtDiff );

        -- enumerated type names all finite-difference
           approximations that are available from BandShell.
```

GLOBAL PROCEDURE AND FUNCTION DEFINITIONS (declaration of the non-standard function clock by setting the alias SystemClock.):

```
function SystemClock : integer;
begin   SystemClock := clock    end;
```

PARAMETERS PASSED TO AND FROM BANDSHELL (must be declared in the

driver program):

1. *n : integer; — specifies the number of equations to be solved, and the number of unknowns.*

2. *jMax : integer; — specifies the number of mesh points.*

3. *ItMax : integer; — specifies the maximum number of iterations to be used by BandCore.*

4. *xMin : RealNumber; — specifies the minimum value of the independent variable in the elliptic domain.*

5. *xMax : RealNumber; — specifies the maximum value of the independent variable in the elliptic domain.*

6. *FactorIncrement : RealNumber; — specifies a factor used to calculate the size of the increment for the numerical differentiation of the CalcEntries routine of BandCore.*

7. *AbsoluteIncrement : RealNumber; — specifies the size of the increment for the numerical differentiation if the increment set with FactorIncrement is less than the SystemZero.*

8. *ImageFirstPoint : boolean; — specifies if the first point is an image point.*

9. *ImageLastPoint : boolean; — specifies if the last point is an image point.*

10. *ReduceTimeOption : boolean; — specifies if a special option should be used to reduce calculation time.*

11. *var Guess : ValueArray; — initial guesses of the values of the dependent variables. (Initial guesses must be set in the main program prior to the BandShell call.)*

12. *var FinalResult : ValueArray; — values of the dependent variables after solution.*

13. *var Deviation : ValueArray; — values of the deviations of the dependent variables from their*

values at the previous iteration.

14. var Residual : ValueArray;
-- values of the equations (residuals) evaluated at the FinalResult.

15. function Equation
(i, j : integer;
x, h : RealNumber;
var NewResult : ValueArray;
function cInterp(k : integer;
x : RealNumber;
var Result : ValueArray) : RealNumber;
function dnFdxn
(n : integer;
function F(x : RealNumber) : RealNumber;
x : RealNumber;
Approx : DiffApprox) : RealNumber
) : RealNumber;

-- specifies the differential equations to be solved, and the boundary conditions.

The function Equation is written in the user's (calling) program, and the specifications of the differential equation therein may include any or all of the following:

-i: integer value of the equation number currently evaluated in BandShell.

-j: integer value of the mesh point currently evaluated in BandShell.

-x: RealNumber representing the location in the domain where the differential equations are currently being evaluated.

-h: RealNumber representing the value of the mesh size.

-NewResult: storage array containing the values of the dependent variables.

-cInterp(k, x, Result): function defined in BandShell that returns the value of the kth dependent variable at a specified location x. cInterp is determined (internally in BandShell) by a linear interpolation between mesh points.

-dnFdxn(n, F, x, Approx): function defined in BandShell that returns the value of the n'th derivative of a function F(x) at a location x. Approx can be any of the DiffApprox options (see global type declarations above).

```
16. function Converged
        ( function x(j : integer) : RealNumber;
          h : RealNumber;
          var NewResult, Deviation : ValueArray;
              var Residual : ValueArray;
          function cInterp( k : integer;
                            x : RealNumber;
                            var Result : ValueArray ) : RealNumber;
          function dnFdxn
                  ( n : integer;
                    function F( x : RealNumber ) : RealNumber;
                    x : RealNumber;
                    Approx : DiffApprox ) : RealNumber
                                          ) : boolean;

                  -- specifies convergence criteria.

                  As in Equation, the specification
                  of the convergence criteria in the
                  user's (calling) program may contain
                  any or all of the following:


                  -x(j): function returns the value of
                         x that is sent to the Equation
                         function at any given mesh point j.


                  -NewResult: array containing the current
                         values of the dependent variables.

                  -Deviation: array containing the current
                         values of the difference between
                         the dependent variables at the current
                         iteration and the previous iteration.

                      -Residual: array containing current
                             values of the equations.

                  -cInterp, dnFdxn, h: defined exactly
                         as in Equation above.


17. procedure NonBandCalcs
            ( LastIteration : boolean;
              Iteration, CPUTime : integer;
              function x(j : integer) : RealNumber;
              h : RealNumber;
              var NewResult, Deviation : ValueArray;
                  var Residual : ValueArray;
              function cInterp( k : integer;
                                x : RealNumber;
                                var Result : ValueArray) : RealNumber;
              function dnFdxn
                    ( n : integer;
                      function F( x : RealNumber ) : RealNumber;
                      x : RealNumber;
                      Approx : DiffApprox    ) : RealNumber );

                  -- indicates additional calculations to
                  be performed with the dependent variables
```

before, during, or after iteration in the BandCore routine.

The specification of the calculations in the user's (calling) program may contain any or all of the following:

-LastIteration: indicates if the system is currently converged ("true" if converged, "false" otherwise).

-Iteration: indicates current iteration number.

-CPUTime: approximate measure of the calculation time used (in milliseconds).

-NewResult: array of the current values of the dependent variables in BandCore.

-Deviation, Residual, x, h, cInterp, dnFdxn: (see above.)

Output: BandShell returns the values of the dependent variables (in the FinalResult array) to the main program for further processing and/or printout.

Procedure and Function Subprogram Hierarchy:

```
-Main (Driver Program)
    --BandShell
        ---SetParameters
        ---CheckParameters
        ---SetMeshSize
        ---SetXDist
        ---Check
            ----WarningMessage
        ---cNodal
        ---jReal
        ---jPrevious
        ---Close
        ---cInterp
        ---dnFdxn
        ---Expression
        ---AttainedConvergence
        ---AuxiliaryManipulations
        ---SetSkipMatrix
            ----ZeroRecord

        ---BandCore

        ---Exit
```

++

...}

```
procedure BandShell
        (
          n, jMax, ItMax : integer;

          xMin, xMax, FactorIncrement, AbsoluteIncrement : RealNumber;

          ImageFirstPoint, ImageLastPoint, ReduceTimeOption : boolean;

          var Guess : ValueArray;

          var FinalResult, Deviation : ValueArray;

            var Residual : ValueArray;


          function Equation
              ( i, j : integer;
                x, h : RealNumber;
                var NewResult : ValueArray;
                function cInterp( k : integer;
                                  x : RealNumber;
                                  var Result : ValueArray        ) : RealNumber;
                function dnFdxn( n : integer;
                                 function F( x : RealNumber ) : RealNumber;
                                 x : RealNumber;
                                 Approx : DiffApprox      ) : RealNumber
                                                            ) : RealNumber;

          function Converged
              ( function x(j : integer) : RealNumber;
                h : RealNumber;
                var NewResult, Deviation : ValueArray;
                  var Residual : ValueArray;
                function cInterp( k : integer;
                                  x : RealNumber;
                                  var Result : ValueArray        ) : RealNumber;
                function dnFdxn( n : integer;
                                 function F( x : RealNumber ) : RealNumber;
                                 x : RealNumber;
                                 Approx : DiffApprox      ) : RealNumber
                                                            ) : boolean;

          procedure NonBandCalcs
              ( LastIteration : boolean;
                Iteration, CPUTime : integer;
                function x(j : integer) : RealNumber;
                h : RealNumber;
                var NewResult, Deviation : ValueArray;
                  var Residual : ValueArray;
                function cInterp( k : integer;
                                  x : RealNumber;
                                  var Result : ValueArray        ) : RealNumber;
                function dnFdxn( n : integer;
                                 function F( x : RealNumber ) : RealNumber;
                                 x : RealNumber;
                                 Approx : DiffApprox) : RealNumber )
                                                                    );
```

```
label     1;        {... used for special exit-condition handler in BandShell }
type      EntryArray = array [1..nLimit, 1..nLimit, 1..jLimit] of boolean;
          SkipCalcRecord = record     A, B, D, X, Y : EntryArray     end;


var       h, Small : RealNumber;
          XDist : array [1..jLimit] of RealNumber;

          Iteration, jCheck, iCheck, ClockInitial, CPUTime : integer;

          CheckActivated, MarkingEntryLocations : boolean;

          Skip, Calc : SkipCalcRecord;

          PrtBandMatrices : array [1..jLimit, 1..ItLimit] of boolean;

           i, j : integer;


procedure Exit;  forward;


procedure SetParameters;

{...      Purpose:  SetParameters sets the value of Small, and sets the values of
                    the PrtBandMatrices array (which is used for debugging
                    of BandCore) to "false".

          Variables global to the routine:  from BandShell --  PrtBandMatrices,
                                                               Small
                                                                          ...}

var       j, Iteration : integer;

begin  { body of SetParameters }

          Small := exp( (-SigDigits/2)*ln(10) );

                                          {... = 10**( -SigDigits/2 ), which
                                            generates a value equal to
                                            the middle range of accuracy
                                            for a real number of order 1.
                                                                          ...}

          for j := 1 to jLimit do
                for Iteration := 1 to ItLimit do
```

```
                    PrtBandMatrices[j,Iteration] := false

end;  { SetParameters }


procedure CheckParameters;

{...    Purpose:  CheckParameters checks the size parameters passed to
                  BandShell to ensure that the size of the user problem does
                  not exceed the limitations of the procedure.  Also, the
                  minimum value of the domain is checked to ensure that it
                  is less than the maximum value.  The procedure causes an
                  Exit if an error is encountered.

          Variables global to the routine:      from BandShell -- n, jMax,
                                                  ItMax, xMax, xMin,
                                                  FactorIncrement, nLimit,
                                                  jLimit, RLimit, Small
                                                                        ...}

type    MessageType = ( NegInt, NTooLarge, TooManyPoints, NotEnoughPoints,
                        TooManyIterations, xMinGTxMax, FacTooSmall,
                                                        FacTooLarge );

var     ErrorFlag : boolean;
        TotalImagePoints : 0..2;


procedure Message( Indicator : MessageType );

{...    Purpose:  Print indicated error messages          ...}

begin
        writeln; writeln(' Warning -- ');

        case Indicator of

        NegInt:         write(' Negative integer parameter encountered ',
                              '( n, jMax or ItMax )');

        NTooLarge:      write(' Number of equations (',n:1,
                              ') is too large (limit = ',nLimit:1,')');

        TooManyPoints:  write(' Number of mesh points (',jMax:1,
                              ') is too large (limit = ',jLimit:1,')');

        NotEnoughPoints: write(' Number of mesh points (',jMax:1,
                              ') is too small (minimum is ',
                              (TotalImagePoints + 1):1,')');

        TooManyIterations: write(' Maximum Number of Iterations (',
                              ItMax:1,') is too Large',
                              ' (Limit = ',ItLimit:1,')');

        xMinGTxMax:     write(' xMin (',xMin:10:5,') is greater than xMax (',
                              xMax:10:5,')');

        FacTooSmall:    write(' FactorIncrement (',FactorIncrement:10,
                              ') is too small (must be greater than ',
```

```
                                Small:5,')');

        FacTooLarge:  write(' FactorIncrement (',FactorIncrement:10,
                            ') is too large (must be less than 1)')


        end;  { Indicator cases }

        writeln; writeln

end;  { Message }



begin  { body of CheckParameters }

        if (ImageFirstPoint) and (ImageLastPoint) then
                TotalImagePoints := 2

        else  if (ImageFirstPoint) or (ImageLastPoint) then
                TotalImagePoints := 1

        else    TotalImagePoints := 0;


        ErrorFlag := false;

        if (n < 0) or (jMax < 0) or (ItMax < 0) then
                begin
                        Message(NegInt);
                        ErrorFlag := true
                end;

        if (n > nLimit) then
                begin
                        Message(NTooLarge);
                        ErrorFlag := true
                end;

        if (jMax > jLimit) then
                begin
                        Message(TooManyPoints);
                        ErrorFlag := true
                end;

        if ( jMax < (TotalImagePoints + 1) )  then
                begin
                        Message(NotEnoughPoints);
                        ErrorFlag := true
                end;

        if (ItMax > ItLimit) then
                begin
                        Message(TooManyIterations);
                        ErrorFlag := true
                end;

        if (xMin > xMax) then
                begin
                        Message(xMinGTxMax);
                        ErrorFlag := true
```

```
            end;

        if (FactorIncrement <= Small) then
            begin
                Message(FacTooSmall);
                ErrorFlag := true
            end;

        if (FactorIncrement > 1) then
            begin
                Message(FacTooLarge);
                ErrorFlag := true
            end;

        if ErrorFlag then Exit

end;  { CheckParameters }



procedure SetMeshSize;

{...    Purpose:  Calculate the mesh size, h.

        Variables global to the routine:  from BandShell --  ImageFirstPoint,
                                                             ImageLastPoint, xMax,
                                                             xMin, jMax, h
                                                                          ...}

var     DomainSize : RealNumber;


begin  { body of SetMeshSize }

        DomainSize := xMax - xMin;

        if (jMax = 1) then
                    h := DomainSize

        else
        if (ImageFirstPoint) and (ImageLastPoint) then
                    h := DomainSize/(jMax - 3)

        else
        if (ImageFirstPoint) and not(ImageLastPoint) then
                    h := DomainSize/(jMax - 2)

        else
        if not(ImageFirstPoint) and (ImageLastPoint) then
                    h := DomainSize/(jMax - 2)

        else
        if not(ImageFirstPoint) and not(ImageLastPoint) then
                    h := DomainSize/(jMax - 1)


end;  { SetMeshSize }
```

```
procedure SetXDist;

{...    Purpose:  Calculate the distance x at each meshpoint of the
                  elliptic domain and store the values in the
                  XDist array.

        Variables global to the routine:  from BandShell -- ImageFirstPoint,
                                                ImageLastPoint, xMax,
                                                xMin, XDist
                                                                    ...}

var     jZero, j : integer;

begin  { body of SetXDist }

        if ImageFirstPoint then
                jZero := 2

        else jZero := 1;


        for j := 1 to jMax do  XDist[j] := (j - jZero)*h + xMin

end;  { SetXDist }




procedure Check( k, j : integer );

{...    Purpose:
                1.      Check the meshpoints to ensure that they are not
                out of range (i. e. that the block-tridiagonal nature of the
                coefficient matrix is preserved).

                2.      If MarkingEntryLocations, then indicate those entries
                of the coefficient matrices of BandCore that must be calculated.
                SetSkipMatrix will use this information to construct a boolean
                array that will indicate to BandCore those calculations which
                may be skipped, thereby saving (in some cases) a considerable
                amount of computational time.

        Variables global to the routine:        from BandShell -- jCheck,
                                                    iCheck, Calc,
                                                    MarkingEntryLocations

                                                                    ...}

procedure WarningMessage;

{...    Purpose:  Print warning messages if a meshpoint is out of range ...}

begin  { body of WarningMessage }

        writeln;
        writeln('WARNING --  Meshpoint location ( = ',j:1,
                ') is out of range for variable ',k:1);
        writeln;
        writeln(' ':10,' in equation ',iCheck:1,
                ' for mesh point number ',jCheck:1,'.');
```

```
        writeln;
        write(' Off-tridiagonal element detected.');
        writeln;
        write(' (Difference approximation may be in error.)');
        writeln; writeln; writeln

end;  { WarningMessage }


begin  { body of Check }

        {... Step 1.  Check to see if meshpoint are out of range ...}

        if ( (j - jCheck) > 2 ) or ( (j - jCheck) < -2 ) then
                WarningMessage

        else      case (j - jCheck) of

                -2:     if (jCheck <> jMax) then WarningMessage;
                -1:     if (jCheck = 1) then WarningMessage;
                 0:     { do nothing };
                 1:     if (jCheck = jMax) then WarningMessage;
                 2:     if (jCheck <> 1) then WarningMessage

                end;  { cases }


        if (j < 1) or (j > jMax) then Exit;


           {... Step 2.  Indicate those matrix entries that will need to
                                be calculated by the CalcEntries routine of
                                BandCore                                ...}

        if MarkingEntryLocations then

        case (j - jCheck) of

       -2:      Calc.Y[iCheck,k,jCheck] := true;
       -1:      Calc.A[iCheck,k,jCheck] := true;
        0:      Calc.B[iCheck,k,jCheck] := true;
        1:      Calc.D[iCheck,k,jCheck] := true;
        2:      Calc.X[iCheck,k,jCheck] := true

        end  { cases }

end;  { Check }



function cNodal( k, j : integer; var Result : ValueArray ) : RealNumber;

{...    Purpose:  Assign the value of variable k at the appropriate
                        meshpoint to cNodal.

        Variables global to the routine:  from BandShell -- CheckActivated
                                                                ...}
```

```
begin  { body of cNodal }
        if CheckActivated then Check(k,j);
        cNodal := Result[k,j]
end;  { cNodal }


function jReal( x : RealNumber ) : RealNumber;
{...    Purpose:  Find the "meshpoint-equivalent" of a distance x.
        Variables global to the routine:        from BandShell -- h, xMin,
                                                    ImageFirstPoint
                                                    ...}
begin  { body of jReal }
        if ImageFirstPoint then
                jReal := (x - xMin)/h + 2
        else    jReal := (x - xMin)/h + 1
end;  { jReal }


function jPrevious( jReal : RealNumber ) : integer;
{...    Purpose:  Use the truncation function to determine the integer value
                  of the closest point BEFORE the location x.  A Small
                  term is added to the value of jReal in order
                  to avoid difficulties with roundoff error.
        Variables global to the routine:        from BandShell -- Small  ...}

begin  { body of jPrevious }
        jPrevious := trunc( jReal + Small )
end;  { jPrevious }
```

```
function Close( jInteger : integer; jReal : RealNumber ) : boolean;

{...    Purpose:  Close is true, if the integer value of the meshpoint location
                  is within a very small increment of the real value of the
                  meshpoint location.

        Variables global to the routine:        from BandShell -- Small
                                                                    ...}


begin  { body of Close }

        if ( abs(jReal - jInteger) < Small ) then
                Close := true

        else    Close := false

end;  { Close }



function cInterp( k : integer; x : RealNumber;
                                var Result : ValueArray ) : RealNumber;

{...    Purpose:  Use a linear interpolation to find the value of the variable k
                  at a given location x within the domain.  If the location
                  x is sufficiently "close" to an actual meshpoint, then the
                  interpolation is not performed.  (This prevents the routine from
                  reaching out of range unnecessarily (see procedure Check for
                  the limitations on the range of meshpoint) ).

        Variables global to the routine:        from BandShell -- h, XDist  ...}


var     xPrevious, cPrevious, cNext, Slope : RealNumber;
        jExact : RealNumber;
        jBefore, jAfter : integer;

begin  { body of cInterp }

        jExact := jReal(x);
        jBefore := jPrevious(jExact);
        jAfter := jBefore + 1;


        if Close( jBefore, jExact ) then
                cInterp := cNodal( k, jBefore, Result )

        else if Close( jAfter, jExact ) then
                cInterp := cNodal( k, jAfter, Result )


        else begin
                cPrevious := cNodal( k, jBefore, Result );
                cNext := cNodal( k, jAfter, Result );
                xPrevious := XDist[jBefore];
```

```
                Slope := ( cNext - cPrevious )/h;

                cInterp := cPrevious + Slope*( x - xPrevious )
            end

end;  { cInterp }
```

```
function dnFdxn( n : integer;
                 function F(x : RealNumber) : RealNumber;
                 x : RealNumber;
                 Approx : DiffApprox              ) : RealNumber;

{...    Purpose:  Calculate nth derivative of F at x (for n = 0, 1, 2).
          (Note: The derivative of order 2 is determined by a recursive
          call to the derivative of order 1.)

          Variables global to the routine:  from BandShell -- h
                                                                    ...}

function ddx( x : RealNumber ) : RealNumber;
begin   ddx := dnFdxn(1,F,x,Approx)      end;


begin  { body of dnFdxn }

        if (n < 0) or (n > 2) then
              writeln(' ':10,'Error -- order of derivative out of range!')
        else

                case n of

                0:      dnFdxn := F(x);


                1:      case Approx of

                    CenDiff :       dnFdxn := ( F(x + h/2) - F(x - h/2) )/h;

                    BackDiff:       dnFdxn := ( F(x) - F(x - h/2) )/(h/2);

                    ForDiff:        dnFdxn := ( F(x + h/2) - F(x) )/(h/2);

                    Back3PtDiff:    dnFdxn := (1.5)*dnFdxn(1,F,x,BackDiff)
                                              - (0.5)*dnFdxn(1,F,x-h,BackDiff);

                    For3PtDiff:     dnFdxn := (1.5)*dnFdxn(1,F,x,ForDiff)
                                              - (0.5)*dnFdxn(1,F,x+h,ForDiff)


                    end;  { Approx cases }


                2:      dnFdxn := dnFdxn(1,ddx,x,Approx)
```

```
            end  { n cases }

end;  { dnFdxn }


function x( j : integer ) : RealNumber;

{...    Purpose:  x determines the appropriate location in the domain that
                  corresponds to a given meshpoint.

        Variables global to the routine:        from BandShell -- XDist
                                                                    ...}

begin   { body of x }

        x := XDist[j]

end;  { x }



function Expression( i, j : integer;
                     var NewResult : ValueArray ) : RealNumber;

{...    Purpose:  Evaluate the user-defined function Equation in order to
                  define the expression of the differential equation that
                  must be solved at each meshpoint according to the protocol
                  required by BandCore.  Set CheckActivated and set the global
                  variables jCheck and iCheck so that warning messages
                  will be printed for values out of range for the
                  block-tridiagonal matrix of BandCore (see function Check).

        Variables global to the routine:        from BandShell -- jCheck,
                                                        iCheck, CheckActivated
                                                                    ...}

begin  { body of Expression }

        CheckActivated := true;

        iCheck := i;
        jCheck := j;


        Expression := Equation( i, j, x(j), h, NewResult,
                                                cInterp, dnFdxn )

end;  { Expression }
```

```
function AttainedConvergence( var NewResult, Deviation : ValueArray;
                              var Residual : ValueArray              ) : boolean;

{...    Purpose: AttainedConvergence sets the user-defined function
                 Converged into the format necessary to be passed to
                 BandCore (as in function Expression above).  It also turns
                 off the CheckActivated flag so that no unnecessary warning
                 messages are printed by the Check function (see function Check).

        Variables global to the routine:        from BandShell -- CheckActivated, h
                                                                                ...}

begin  { body of AttainedConvergence }

       CheckActivated := false;

       AttainedConvergence := Converged( x, h, NewResult, Deviation,
                                         Residual, cInterp, dnFdxn )

end;  { AttainedConvergence }



procedure AuxiliaryManipulations( LastIteration : boolean;
                                  Iteration : integer;
                                      var NewResult, Deviation : ValueArray;
                                      var Residual : ValueArray );

{...    Purpose: Call the user-defined routine NonBandCalcs.  Also, turn
                 off the CheckActivated flag (see AttainedConvergence above).

        Variables global to the routine:        from BandShell -- CheckActivated,
                                                           ClockInitial, h
                                                                        ...}


var     CPUTime : integer;

begin  { body of AuxiliaryManipulations }

       CheckActivated := false;

       CPUTime := SystemClock - ClockInitial;

       NonBandCalcs( LastIteration, Iteration, CPUTime,
                     x, h, NewResult, Deviation, Residual, cInterp, dnFdxn )

end;  { AuxiliaryManipulations }



procedure SetSkipMatrix;

{...    Purpose: If ReduceTimeOption is true, call the function Expression
                 (with Guess) for all possible equations i and
                 meshpoints j with the MarkingEntryLocations flag set equal
```

```
                    to "true".  The Check routine will indicate all
                    matrix entries that must be calculated by the CalcEntries
                    routine of BandCore by indicating the value
                    of "true" at the appropriate location of the record
                    Calc.  The record Skip, then, has the characteristic
                    of not(Calc).

                    This procedure permits the CalcEntries routine of
                    BandCore to eliminate some unnecessary
                    calculations.  (Specifically, it eliminates the need to
                    calculate the numerical derivative of Expression
                    for those cases in which it will always be zero.)

                If ReduceTimeOption is false, set all entry locations of
                    Skip to "false".

        Variables global to the routine:        from BandShell -- ReduceTimeOption,
                                                    Guess,
                                                    MarkingEntryLocations,
                                                    iCheck, jCheck,
                                                    CheckActivated
                                                                        ...}

var     i, k, j : integer;
        Dummy : RealNumber;


procedure ZeroRecord( var SkipOrCalc : SkipCalcRecord );

{...    Purpose:  Set all entries of SkipOrCalc equal to false    ...}

var     i, k, j : integer;

begin  { body of ZeroRecord }

        with SkipOrCalc do

        for j := 1 to jMax do
            for i := 1 to n do
                for k := 1 to n do
                begin
                        A[i,k,j] := false;
                        B[i,k,j] := false;
                        D[i,k,j] := false;
                        X[i,k,j] := false;
                        Y[i,k,j] := false
                end

end;  { ZeroRecord }



begin  { body of SetSkipMatrix }

        case ReduceTimeOption of

        true :  begin
```

```
                    ZeroRecord(Calc);

                    MarkingEntryLocations := true;
                    CheckActivated := true;


                    for j := 1 to jMax do
                        for i := 1 to n do
                        begin

                            iCheck := i;
                            jCheck := j;

                            Dummy := Expression( iCheck, jCheck, Guess )

                        end;

                    MarkingEntryLocations := false;
                    CheckActivated := false;


                    for j := 1 to jMax do
                        for i := 1 to n do
                            for k := 1 to n do
                            begin
                                Skip.A[i,k,j] := not(Calc.A[i,k,j]);
                                Skip.B[i,k,j] := not(Calc.B[i,k,j]);
                                Skip.D[i,k,j] := not(Calc.D[i,k,j]);
                                Skip.X[i,k,j] := not(Calc.X[i,k,j]);
                                Skip.Y[i,k,j] := not(Calc.Y[i,k,j])
                            end


                end; { true }

        false:  begin

                    ZeroRecord(Skip);

                    MarkingEntryLocations := false;

                end { false }

        end { ReduceTimeOption cases }

end; { SetSkipMatrix }
```

{...

++

Procedure Title: BandCore -- Version 3, Release 3.2

Written By: Michael Matlosz
Date: October 20, 1984

Procedure Summary: BandCore facilitates the solution of coupled, non-linear, difference equations. Although it is fairly general, the routine is typically used to solve (by finite-difference techniques) boundary-value problems consisting of systems of second-order, coupled, non-linear, ordinary differential equations. The technique employed is that of Newman, and the ArrayManipulation, GaussElimination, and BackSubstitution routines are based on the algorithms of Newman. ArrayManipulation comprises the major part of the algorithm used for Newman's subroutine BAND(J), BackSubstitution comprises the remainder of BAND(J), and GaussElimination consists of the algorithm of Newman's MATINV(N,M).

BandCore is quite general since the matrix entries needed for the ArrayManipulation routine are determined by a simple numerical differentiation. This numerical differentiation is contained in the CalcEntries routine. For details of the method, the following references are suggested:

John Newman, "Numerical Solution of Coupled, Ordinary Differential Equations (UCRL-17739)," Lawrence Radiation Laboratory, University of California, Berkeley, August 1967.

John Newman, "Numerical Solution of Coupled, Ordinary Differential Equations," IND. ENG. CHEM. FUNDAMENTALS 7, 514-17 (1968)

John Newman, "Electrochemical Systems," (Englewood Cliffs, NJ: Prentice Hall, Inc., 1973). Appendix C, pp. 414-25.

Ralph White, Charles M. Mohr, Peter Fedkiw, and John Newman, "The Fluid Motion Generated by a Rotating Disk: A Comparison of Solution Techniques," LBL-3910, Lawrence Berkeley Laboratory, November 1975.

Input: *The procedure is intended to be used in conjunction with a driver program, BandShell, which specifies the parameters. A summary of these parameters is given below, with a short explanation of the function of each.*

IDENTIFIERS GLOBAL TO BANDCORE :

1. *"const" declarations:*

```
nLimit = (integer);  -- indicates the maximum number
                        of equations and unknowns.

jLimit = (integer);  -- indicates the maximum number
                        of meshpoints.

ItLimit = (integer);  -- indicates the maximum limit
                         on the number of iterations
                         allowed.

WidthLimit = (integer)  -- indicates the maximum column
        width of matrices in ArrayManipulation.  (WidthLimit
        should be at least 2*nLimit + 1.)

SystemZero = (RealNumber);  -- sets the smallest (absolute)
                               value for a RealNumber.
```

2. *"type" declarations:*

```
RealNumber = real;  OR  RealNumber = double;
                    OR  RealNumber = quad;
        --  type of the real numbers. Set in the
        calling routine to be one of the following:
                real   ( 7 digit accuracy )
                double  ( 15 digit accuracy )
                quad   ( 33 digit accuracy ).

ValueArray = array [1..nLimit, 1..jLimit] of RealNumber;
        --  storage-array type to contain the values of
        the dependent variables.
```

3. *"var" declarations (Note: values must be assigned to these variables in the calling routine. They should not be left undefined!):*

```
Skip : record
          A, B, D, X, Y :
                array [1..nLimit,1..nLimit,1..jLimit]
                                                of  boolean
       end;
        -- record of arrays contains boolean elements that
        indicate if some of the calculations of BandCore
        can be avoided to save time.

PrtBandMatrices : array [1..jLimit, 1..ItLimit] of boolean;
          -- array contains boolean elements that
        indicate if the matrices A, B, D, X, Y and the
        vector G should be printed at specified meshpoints
        (and iterations).
```

PARAMETERS PASSED TO BANDCORE :

1. *n : integer -- specifies the number of equations to be solved.*

2. *jMax : integer -- specifies the number of meshpoints.*

3. *RMax : integer -- specifies the maximum number of iterations allowed.*

4. *FactorIncrement : RealNumber -- used to calculate the size of the increment used in the numerical differentiation.*

5. *AbsoluteIncrement : RealNumber -- indicates the size of the increment used for the numerical differentiation if the value of the increment calculated with FactorIncrement is less than the SystemZero.*

6. *function Expression(i, j : integer;*
   var NewResult : ValueArray) : RealNumber;
   -- This function specifies the finite-difference expressions to be solved. (The expressions are given in terms of the NewResult.)

7. *function AttainedConvergence*
   (var NewResult, Deviation : ValueArray;
   var Residual : ValueArray) : boolean;
   -- This function specifies the convergence criteria. (The criteria are given in terms of the NewResult, Deviation and/or Residual.)

8. *procedure AuxiliaryManipulations*
   (LastIteration : boolean;
   Iteration : integer;
   var NewResult, Deviation : ValueArray;
   var Residual : ValueArray);
   -- This procedure specifies additional calculations to be performed during the execution of BandCore.

9. *var Guess : ValueArray;*
   -- values of the initial guesses for the dependent variables. (Note: they must be supplied by the calling routine.)

10. *var FinalResult : ValueArray;*
    -- values of the dependent variables. When BandCore is completed, FinalResult contains the converged values of the dependent variables.

11. *var Deviation : ValueArray;*
    -- deviations in the dependent variables from the previous iteration.

12. *var Residual : ValueArray;*
    -- values of the Expression at the converged values of the dependent variables.

13. *var Iteration : integer;*

-- This variable contains the number of iterations used. It is passed back to the calling routine after processing.

Output: *BandCore returns the values of the dependent variables (in the FinalResult array) to the calling program for further processing or printout.*

Procedure and Function Subprogram Hierarchy:

```
-Calling Routine (BandShell)
      --BandCore
            ---ErrorDiagnostic
            ---PrintEntries
                  -----PrintMatrix
                  -----PrintVector
            ---Exit
            ---ZeroEntries
            ---ResetABDXY
            ---ResetG
            ---CalcEntries
                  -----NumericalDerivative
            ---GaussElimination
                  -----Search
            ---AW,AE,YE,QE,AZ,YZ,QZ,EC,WC
            ---ArrayManipulation
            ---BackSubstitution
```

++

...}

```
procedure BandCore( n, jMax, ItMax : integer;

                    FactorIncrement, AbsoluteIncrement : RealNumber;

                    function Expression( i, j : integer;
                                         var NewResult : ValueArray) : RealNumber;

                    function AttainedConvergence
                                  ( var NewResult, Deviation : ValueArray;
                                    var Residual : ValueArray         ) : boolean;

                    procedure AuxiliaryManipulations
                              ( LastIteration : boolean;
                                Iteration : integer;
                                     var NewResult, Deviation : ValueArray;
                                     var Residual : ValueArray );

                    var Guess : ValueArray;

                    var FinalResult, Deviation : ValueArray;
```

```
                    var Residual : ValueArray;
                    var Iteration : integer                    );

label    1;        {... used for special exit-condition handler in BandCore }
const    RowSize = nLimit;

type     EntryType = record A,B,D,X,Y,G : RealNumber    end;
         Vector = array [1..nLimit] of RealNumber;
         Matrix = array [1..nLimit, 1..nLimit] of RealNumber;
         AugMatrix = array [1..nLimit, 1..WidthLimit] of RealNumber;

var      i, k, j : integer;
         NewResult : ValueArray;
         A, B, D, X, Y, Q, W : Matrix;
         E : array [1..nLimit, 1..nLimit, 1..jLimit] of RealNumber;
         G : Vector;
         C, Z : array [1..nLimit, 1..jLimit] of RealNumber;
         LastIteration, ZeroDetermFlag : boolean;


procedure ErrorDiagnostic( Message : integer ); forward;
procedure PrintEntries( j, Iteration : integer ); forward;
procedure Exit;  forward;


procedure ZeroEntries;
{...     Purpose:  Rezero all storage locations for matrix entries of ArrayManipulation
         Variables global to the routine:  from BandCore --  A,B,D,X,Y,G,n
                                                                    ...}
var  i, k : integer;
begin
         for i := 1 to n do
         begin
```

```
                G[i] := 0;

                for k := 1 to n do begin

                        A[i,k] := 0;
                        B[i,k] := 0;
                        D[i,k] := 0;
                        X[i,k] := 0;
                        Y[i,k] := 0

                end
        end

end; { ZeroEntries }


procedure ResetABDXY( Entry : EntryType; i, k : integer );

{...    Purpose :  Adds to the appropriate storage locations the
                   contributions to the A, B, D, X and Y matrix
                   entries of ArrayManipulation.

        Variables global to the routine :  from BandCore -- A,B,D,X,Y
                                                                  ...}

begin

        A[i,k] := A[i,k] + Entry.A;
        B[i,k] := B[i,k] + Entry.B;
        D[i,k] := D[i,k] + Entry.D;
        X[i,k] := X[i,k] + Entry.X;
        Y[i,k] := Y[i,k] + Entry.Y

end; { ResetEntries }


procedure ResetG( Entry : EntryType; i : integer );

{...    Purpose: Adds to the appropriate storage location
                 the contribution to the G vector of ArrayManipulation.

        Variables global to the routine: from BandCore -- G
                                                          ...}

begin

        G[i] := G[i] + Entry.G

end; { ResetG }


procedure CalcEntries( i : integer );

{... Purpose: Calculate a row of entries for the matrices A, B, D, X, and Y
              and calculate a single entry for the vector G, for a given
```

```
                equation.

        Variables global to the routine:  from BandCore -- NewResult
                                                                ...}

type    Subscript = (Zero, ZeroPlusEpsilon);

var     k : integer;
        Entry : EntryType;
        F : array [Subscript] of RealNumber;


function NumericalDerivative( i, k, jSpec : integer ) : RealNumber;

{...  Purpose:  Determine the derivative of the i'th Expression with
                respect to the variable k at the mesh point specified.
                The derivative is calculated numerically by a forward-
                difference formula.

                The derivative is the coefficient of the first term
                of a Taylor-series expansion of the Expression about
                the current values of the variables.  The matrix entries
                for A, B, D, X and Y are directly related to
                the coefficients, as shown in the body of CalcEntries.

        Variables global to the routine:        from BandCore -- FactorIncrement,
                                                          AbsoluteIncrement,
                                                          NewResult
                                                                ...}

var     Factor, Epsilon, AbsVarValue, SavedValue : RealNumber;


begin { body of NumericalDerivative }

        Factor := FactorIncrement;


                {... Set the increment (Epsilon) according to the
                        absolute magnitude of the value of the variable k
                        (at the meshpoint specified)                ...}

        AbsVarValue := abs( NewResult[k,jSpec] );


        Epsilon := Factor*AbsVarValue;

        if (Epsilon < SystemZero) then
                Epsilon := AbsoluteIncrement;


                {... Increment the specific value of the NewResult,
```

```
                    making sure to save the original value.          ...}
        SavedValue := NewResult[k,jSpec];
        NewResult[k,jSpec] := NewResult[k,jSpec] + Epsilon;

                {... Calculate the expression with the incremented values ...}
        F[ZeroPlusEpsilon] := Expression( i, j, NewResult );
                {... Return values to their original condition ...}
        NewResult[k,jSpec] := SavedValue;


                {... Calculate the derivative ...}
        NumericalDerivative := ( F[ZeroPlusEpsilon] - F[Zero] )/Epsilon

end; { NumericalDerivative }


begin { body of CalcEntries }
        F[Zero] := Expression( i, j, NewResult ); {... Calculate the expression
                                                    with the current values ...}

        for k := 1 to n do
        begin
                if (j <= 2) or (Skip.Y[i,k,j]) then
                        Entry.Y := 0
                else Entry.Y := NumericalDerivative(i,k,j-2);

                if (j = 1) or Skip.A[i,k,j] then
                        Entry.A := 0
                else Entry.A := NumericalDerivative(i,k,j-1);

                if Skip.B[i,k,j] then
                        Entry.B := 0
                else Entry.B := NumericalDerivative(i,k,j);

                if (j = jMax) or Skip.D[i,k,j] then
                        Entry.D := 0
                else Entry.D := NumericalDerivative(i,k,j+1);

                if (j >= jMax-1) or Skip.X[i,k,j] then
                        Entry.X := 0
                else Entry.X := NumericalDerivative(i,k,j+2);


                ResetABDXY(Entry,i,k)

        end;
```

```
        Entry.G := -F[Zero];
        ResetG(Entry,1)
end;  { CalcEntries }


procedure GaussElimination( R : Matrix;
                            var S : AugMatrix;
                            T : AugMatrix;
                            n, m : integer        );

{...    Purpose:  Determine the solution to the matrix equation R*S = T
                  (R is n by n, and S,T are n by m).  R,T,n and m are sent
                  to the routine as parameters; S, the solution matrix, is
                  returned to the calling routine.

        Method:  Gaussian Elimination using elementary row operations.
                 For each of the n row elimination steps, the following
                 four steps are repeated:

                 Step 1:  Determination of the pivot.  Go through the
                 rows of R, one at a time (skipping rows already used),
                 in order to determine the location of the largest entry
                 (in absolute value) in the row with the smallest ratio
                 of second-largest entry to largest entry (i. e., the
                 smallest ratio NextToMaxEntry/MaxEntry).  Thus, the
                 BestMaxEntry is the largest entry in the row with the
                 BestRatio.  This BestMaxEntry will become the Pivot.
                 (This choice of pivot reduces roundoff error.)

                 Step 2:  Row interchange.  If the BestMaxEntry (the choice
                 for pivot) is not on the diagonal of R, then two rows
                 are interchanged such that BestMaxEntry is on the diagonal.

                 Step 3:  Division by Pivot.  BestMaxEntry becomes the
                 Pivot, and each element of the row containing the Pivot
                 is divided by Pivot.  (The diagonal entry of this row
                 of R is now unity.)

                 Step 4:  Elimination.  All entries in the column containing the
                 Pivot (except the Pivot itself) are eliminated by suitable
                 row multiplications and subtractions.

        Variables global to the routine:  from BandCore -- ZeroDetermFlag

                                                                    ...}

label     2;        {... label for exit-condition handler for zero-determinant }

var   Row, Column, PivotColumn, PivotRow, RowChoice : integer;
      NumberOfRowEliminations, ColWithMaxRowEntry : integer;

      UsedRow, UsedCol : array [1..RowSize] of boolean;

      MaxRowEntry, NextToMaxEntry : RealNumber;
      PresentRatio, BestRatio : RealNumber;
      Multiplier, Pivot, SavedValue : RealNumber;
```

```
      DetermIsZero : boolean;

procedure Search( Row : integer );

{... Purpose: Search through a row of matrix R to find the largest
              entry of the row (MaxRowEntry) and the second-largest
              entry of the row (NextToMaxEntry). Also, indicate the
              column containing the MaxRowEntry, and activate the
              DetermIsZero flag if the row contains only zeros.

     Variables global to the routine: from GaussElimination --
                                        R, n, ColWithMaxRowEntry,
                                        MaxRowEntry, NextToMaxEntry,
                                        UsedCol, DetermIsZero
                                                                ...}

var   Column : integer;

begin { body of Search }

      MaxRowEntry := 0;  NextToMaxEntry := 0;

      for Column := 1 to n do
          if not UsedCol[Column] then

            if abs(R[Row,Column]) > MaxRowEntry then
                    begin
                        NextToMaxEntry := MaxRowEntry;
                        MaxRowEntry := abs(R[Row,Column]);
                        ColWithMaxRowEntry := Column
                    end

                else if abs(R[Row,Column]) > NextToMaxEntry then

                        NextToMaxEntry := abs(R[Row,Column]);


      if ( MaxRowEntry = 0 ) then DetermIsZero := true

end; { Search }


begin { body of GaussElimination }

   {... Initializations ...}

      S := T;

      DetermIsZero := false;

      for Row := 1 to n do UsedRow[Row] := false;
      for Column := 1 to n do UsedCol[Column] := false;

   {... Solve the equations ...}
```

```
for NumberOfRowEliminations := 1 to n do

begin  { row eliminations }

    {... Step 1:  Pivot Determination ...}

          BestRatio := 1.1;  {... setting BestRatio to 1.1 guarantees
                                  that the test "if PresentRatio < BestRatio"
                                  below will succeed on the first pass
                                                                    ...}

        for Row := 1 to n do
        if not UsedRow[Row] then
        begin
                Search(Row);
                if DetermIsZero then goto 2;

                PresentRatio := NextToMaxEntry/MaxRowEntry;

                if PresentRatio < BestRatio then
                    begin
                        BestRatio := PresentRatio;

                        RowChoice := Row;
                        PivotColumn := ColWithMaxRowEntry
                    end

        end;

        PivotRow := PivotColumn;
        UsedCol[PivotColumn] := true;


    {... Step 2:  Row Interchange ...}

        if RowChoice <> PivotRow then
        begin
                for Column := 1 to n do
                begin
                        SavedValue := R[RowChoice,Column];
                        R[RowChoice,Column] := R[PivotRow,Column];
                        R[PivotRow,Column] := SavedValue
                end;

                for Column := 1 to m do
                begin
                        SavedValue := S[RowChoice,Column];
                        S[RowChoice,Column] := S[PivotRow,Column];
                        S[PivotRow,Column] := SavedValue
                end
        end;

        UsedRow[PivotRow] := true;


    {... Step 3:  Divide by Pivot ...}

        Pivot := R[PivotRow,PivotColumn];

        for Column := 1 to n do R[PivotRow,Column] :=
                                        R[PivotRow,Column]/Pivot;
        for Column := 1 to m do S[PivotRow,Column] :=
```

```
                                        S[PivotRow,Column]/Pivot;

        {... Step 4:  Elimination ...}

            for Row := 1 to n do
            if Row <> PivotRow then
            begin
                    Multiplier := R[Row,PivotColumn];
                    for Column := 1 to n do R[Row,Column] := R[Row,Column]
                                    - Multiplier*R[PivotRow,Column];
                    for Column := 1 to m do S[Row,Column] := S[Row,Column]
                                    - Multiplier*S[PivotRow,Column]
            end;


    end;  { row eliminations }


2:  if DetermIsZero then
            begin
                    ErrorDiagnostic(1);
                    PrintEntries(j,Iteration);
                    ZeroDetermFlag := true
            end


end;  { GaussElimination }




{...
        Matrix-Multiplication Declarations
        Purpose:  The following nine function declarations are used
                as a shorthand notation in procedures ArrayManipulation and
                BackSubstitution.  Each defines a unique matrix product,
                which is used in one of those procedures.

        Variables global to the routines:  from BandCore --  A,C,E,Q,W,Y,Z,n
                                                                    ...}


function AW( k, m : integer ) : RealNumber;
var  index : integer; Sum : RealNumber;
begin   Sum := 0;
        for index := 1 to n do Sum := Sum + A[k,index]*W[index,m];
        AW := Sum  end;

function AE( i, k, j : integer ) : RealNumber;
var  index : integer; Sum : RealNumber;
begin   Sum := 0;
        for index := 1 to n do Sum := Sum + A[i,index]*E[index,k,j];
        AE := Sum  end;

function YE( i, l, j : integer ) : RealNumber;
var  index : integer; Sum : RealNumber;
begin   Sum := 0;
        for index := 1 to n do Sum := Sum + Y[i,index]*E[index,l,j];
```

```
        YE := Sum  end;

function QE( i, k, j : integer ) : RealNumber;
var  index : integer; Sum : RealNumber;
begin   Sum := 0;
        for index := 1 to n do Sum := Sum + Q[i,index]*E[index,k,j];
        QE := Sum  end;

function AZ( i, j : integer ) : RealNumber;
var  index : integer; Sum : RealNumber;
begin   Sum := 0;
        for index := 1 to n do Sum := Sum + A[i,index]*Z[index,j];
        AZ := Sum  end;

function YZ( i, j : integer ) : RealNumber;
var  index : integer; Sum : RealNumber;
begin   Sum := 0;
        for index := 1 to n do Sum := Sum + Y[i,index]*Z[index,j];
        YZ := Sum  end;

function QZ( i, j : integer ) : RealNumber;
var  index : integer; Sum : RealNumber;
begin   Sum := 0;
        for index := 1 to n do Sum := Sum + Q[i,index]*Z[index,j];
        QZ := Sum  end;

function EC( k, j1, j2 : integer ) : RealNumber;
var  index : integer; Sum : RealNumber;
begin   Sum := 0;
        for index := 1 to n do  Sum := Sum + E[k,index,j1]*C[index,j2];
        EC := Sum  end;

function WC( k, j : integer ) : RealNumber;
var  index : integer; Sum : RealNumber;
begin   Sum := 0;
        for index := 1 to n do  Sum := Sum + W[k,index]*C[index,j];
        WC := Sum  end;


{... End Matrix-Multiplication Declarations ...}



procedure ArrayManipulation( j : integer );

{... Purpose: Determination of the decomposition matrices E and Z
               at each meshpoint.

     Method: ArrayManipulation uses the matrices A, B, D, X, Y
             and the vector G at each meshpoint in order to determine
             the appropriate entries of matrices R and T of an
             equation of the form R*S = T.  (At each meshpoint,
             the solution of an equation of this form will yield
             the decomposition matrices E and Z, which are used by
             BackSubstitution to determine the solution of the
             finite-difference equations.) ArrayManipulation calls
             the routine GaussElimination(R,S,T,n,m) and recovers
```

the solution matrix S from which ArrayManipulation determines the matrices E and Z. (Addition: at meshpoint 1, a matrix W is generated, which accounts for the presence of an off-tridiagonal element at point 1. This is a special case, and is important if the boundary conditions at point 1 contain a derivative that requires a three-point difference formula. This condition will exist whether or not image points are used at that boundary.)

At each meshpoint, the matrices R and T are determined from A, B, D, X, Y and G by a number of algebraic equations. The form of these equations comes from a decomposition of the block-tridiagonal matrix of matrices that is formed by the coefficients of the dependent variables in the finite-difference approximation to the differential equations. This decomposition process is an extension of the Thomas method for the solution of block-tridiagonal matrices containing single elements (see White, et al. for a more complete explanation). These decomposition equations are the heart of the method, and they are listed in the appendix of John Newman's book (see references at the start of BandCore). The purpose of ArrayManipulation is simply to determine R and T from these equations, call GaussElimination, and assign the solution matrix S to the appropriate storage location (E or Z).

In the listing given here, after each expression in the procedure, the corresponding equation from Appendix C of John Newman's book is given. Unfortunately, due to the limitations of the character set, not all of the notation is the same. A short comparison of the major notation is provided below:

Appendix C of "Electrochemical Systems"	*This procedure*
A[i,k]	*A[i,k]*
B[i,k]	*B[i,k]*
C[k,j]	*C[k,j]*
D[i,k]	*D[i,k]*
E[k,l,j]	*E[k,l,j]*
G[i]	*G[i]*
X[i,k]	*X[i,k]*
Y[i,k]	*Y[i,k]*
a[i,l] (lower case a)	*Q[i,l]*
b[i,k] (lower case b on page 418)	*R[i,k]*
x[k,l] (lower case x)	*W[k,l]*
xi[k,j] (greek letter xi)	*Z[k,j]*
j	*j*
j[max]	*jMax*

Variables global to the routine: from BandCore -- A,B,D,X,Y,G,Q, W,E,Z,n,jMax

...}

```
var     i, k, l, m : integer;

        R : Matrix;
        S, T : AugMatrix;
```

```
begin  { body of ArrayManipulation }
        if (j=1) then  { case j of 1 }
        begin
                for i := 1 to n do
                    for k := 1 to n do R[i,k] := B[i,k];
                                                {... define coefficient of Z,
                                                  E, and W on LHS of (C-15) }
                for i := 1 to n do
                    for l := 1 to n do  T[i,l] := -D[i,l];
                                                {... define RHS for 2nd section
                                                  of (C-15) }
                for i := 1 to n do  T[i,n + 1] := G[i];
                                                {... define RHS for 1st section
                                                  of (C-15) }
                for i := 1 to n do
                   for l := 1 to n do  T[i,n + 1 + l] := -X[i,l];
                                                {... define RHS for 3rd section
                                                  of (C-15) }

                GaussElimination(R,S,T,n,2*n+1);

                for k := 1 to n do
                    for l := 1 to n do  E[k,l,j] := S[k,l];
                                                {... assign solution of 2nd
                                                  section of (C-15) }
                for k := 1 to n do  Z[k,j] := S[k,n + 1];
                                                {... assign solution of 1st
                                                  section of (C-15) }
                for k := 1 to n do
                        for l := 1 to n do W[k,l] := S[k,n + 1 + l]
                                                {... assign solution of 3rd
                                                  section of (C-15) }
           end  { j = 1 }

        else if (j=2) then  { case j of 2 }
        begin
                for i := 1 to n do
                    for k := 1 to n do R[i,k] := B[i,k] + AE(i,k,j-1);
                                        {... define coefficient of E on LHS
                                          of (C-18) ds defined by (C-19) }
                for k := 1 to n do
                    for m := 1 to n do T[k,m] := -D[k,m] - AW(k,m);
                                        {... slightly different form of RHS
                                          of (C-18) defined if j=2 }
```

```
        for i := 1 to n do  T[i,n + 1] := G[i] - AZ(i,j-1);
                                                {... define RHS of (C-17) }

        GaussElimination(R,S,T,n,n+1);

        for k := 1 to n do
            for m := 1 to n do  E[k,m,j] := S[k,m];
                                                {... assign solution of
                                                   (C-18) to E }

        for k := 1 to n do  Z[k,j] := S[k,n + 1]
                                                {... assign solution of
                                                   (C-17) to Z }
end  { j = 2 }


else if (j>2) and (j<jMax) then
                                { case of j > 2 and j < jMax }

begin

    for i := 1 to n do
        for k := 1 to n do R[i,k] := B[i,k] + AE(i,k,j-1);
                                {... define coefficient of E on LHS
                                   of (C-17) and (C-18) as given by
                                   (C-19) }

    for i := 1 to n do
        for m := 1 to n do T[i,m] := -D[i,m];
                                {... define RHS of (C-18) }


    for i := 1 to n do T[i,n + 1] := G[i] - AZ(i,j-1);
                                                {... define RHS of (C-17) }

    GaussElimination(R,S,T,n,n+1);

    for k := 1 to n do
        for m := 1 to n do E[k,m,j] := S[k,m];
                                                {... assign solution of (C-18)
                                                      to E }
    for k := 1 to n do Z[k,j] := S[k,n + 1]
                                                {... assign solution of (C-17)
                                                      to Z }

end  { case j > 2 and  j < jMax }


else if (j=jMax) then  { case j of jMax }

begin

        for i := 1 to n do
            for l := 1 to n do  Q[i,l] := A[i,l]
                                        + YE(i,l,j-2);
                                                {... define Q in
                                                   1st part of (C-21) }

        for i := 1 to n do T[i,1] := G[i] - YZ(i,j-2)
                                    - QZ(i,j-1);
```

```
                                                    {... define RHS of (C-20) }
            for i := 1 to n do
                for k := 1 to n do  R[i,k] := B[i,k]
                                              + QE(i,k,j-1);
                                                    {... define coefficient
                                                      of C on LHS of (C-20) }
            GaussElimination(R,S,T,n,1);

            for k := 1 to n do Z[k,j] := S[k,1]
                                                    {... assign solution of
                                                      (C-20) to Z.  The value
                                                      will be transferred to
                                                      C in the first step of
                                                      BackSubstitution }

        end  { j = jMax }

end;  { ArrayManipulation }


procedure BackSubstitution;

{... Purpose:  Use the values of E, Z and W determined by ArrayManipulation,
               in conjunction with equations (C-14), (C-16), and (C-20), to
               assign the results to the deviation variables C.

        Variables global to the routine:  from BandCore --- C,E,W,Z,n,jMax
                                                                        ...}

var  k, j : integer;


begin  { body of BackSubstitution }

        for k := 1 to n do C[k,jMax] := Z[k,jMax];
                                                    {... assign solution
                                                      of (C-20) to C.  It
                                                      has been stored in
                                                      Z after assignment in
                                                      the ArrayManipulation
                                                      routine }

        if (jMax > 1) then
        begin

            for j := jMax - 1 downto 2 do
                for k := 1 to n do  C[k,j] := Z[k,j]
                                              + EC(k,j,j+1);
                                                    {... assign solution
                                                      to C on LHS of (C-16) }


            for k := 1 to n do  C[k,1] := Z[k,1] + EC(k,1,2) + WC(k,3)
                                                    {... assign solution
                                                      to C on LHS of (C-14) }

        end
```

```
end;  { BackSubstitution }


procedure ErrorDiagnostic;

{...    Purpose:  Print a specified error message.

        Variables global to the routine:  from BandCore -- j, Iteration
                                                                    ...}
begin
        case Message of

        1:  begin
                writeln; writeln;
                write(' Determinant is Zero at meshpoint = ',j:3);
                write('    Iteration = ',Iteration:2);
                writeln
            end;  { 1 }

        2:  begin
                writeln; writeln;
                write(' WARNING -- System did not converge in the',
                                                ItMax:3);
                if (ItMax=1) then write(' iteration allowed')
                        else write(' iterations allowed');
                writeln
            end;  { 2 }

        3:  begin
                writeln; writeln;
                write(' WARNING -- Number of equations and unknowns too large.',
                        'Matrix width limit exceeded!');
                writeln
            end;  { 3 }

        4:  begin
                writeln; writeln;
                write(' WARNING -- WidthLimit not compatible with nLimit.',
                        '(WidthLimit must be at least 2*nLimit + 1.)');
                writeln
            end  { 4 }

        end  { Message cases }

end;  { ErrorDiagnostic }


procedure PrintEntries;

{...    Purpose:  Diagnostic routine used to print the matrix entries
                  of ArrayManipulation at a specified meshpoint
                  and Iteration

        Variables global to the routine:  from BandCore -- A,B,D,X,Y,G,n
                                                                    ...}
```

```
procedure  PrintMatrix( M : Matrix; MName : char );
var     Row, Col : integer;
begin
        writeln; writeln; write(' ':5,' Matrix ',MName,' -- ');
        writeln;
        for Row := 1 to n do
        begin
                writeln; writeln;
                for Col := 1 to n do
                        write(' ':5,M[Row,Col]:12)
        end;
        writeln; writeln
end;  { PrintMatrix }

procedure PrintVector( V : Vector; VName : char );
var     Row : integer;
begin
        writeln; writeln; write(' ':5,' Vector ',VName,' -- ');
        writeln;
        for Row := 1 to n do
        begin
                writeln; writeln;
                write(' ':10,V[Row]:12);
        end;
        writeln; writeln;
end;  { PrintVector }


begin
        writeln; writeln;
        write('Debug Diagnostic -- Matrix Entry Listing');
        writeln;
        write('Meshpoint = ',j:3,'      Iteration = ',Iteration:2); writeln;
        PrintMatrix(A,'A');
        PrintMatrix(B,'B');
        PrintMatrix(D,'D');
        PrintMatrix(X,'X');
        PrintMatrix(Y,'Y');
        PrintVector(G,'G');
        writeln; writeln; writeln; writeln; writeln
end;



procedure Exit;
begin
        writeln;
        writeln('BANDCORE EXECUTION INTERRUPTED');
        goto 1     {... break and go to BandCore exit label ...}
end;  { Exit }



begin  { body of BandCore }

        if ( WidthLimit < (2*n + 1) ) then ErrorDiagnostic(3);
        if ( WidthLimit < (2*nLimit + 1) ) then ErrorDiagnostic(4);

        NewResult := Guess;
```

```
Iteration := 0;  ZeroDetermFlag := false;  LastIteration := false;

AuxiliaryManipulations( LastIteration, Iteration,
                                NewResult, Deviation, Residual );

repeat
        Iteration := Iteration + 1;

        for j := 1 to jMax do begin

                ZeroEntries;

                for i := 1 to n do
                begin
                        CalcEntries(i);
                end;

                if PrtBandMatrices[j,Iteration] then PrintEntries(j,Iteration);

                ArrayManipulation(j)

        end;

        BackSubstitution;

        for j := 1 to jMax do
            for k := 1 to n do
            begin

                Deviation[k,j] := C[k,j];

                NewResult[k,j] := NewResult[k,j] + Deviation[k,j]

            end;


          for j := 1 to jMax do
              for i := 1 to n do
                  Residual[i,j] := Expression( i, j, NewResult );


        AuxiliaryManipulations( LastIteration, Iteration,
                                        NewResult, Deviation, Residual );

        if ZeroDetermFlag then Exit

until AttainedConvergence(NewResult, Deviation, Residual)
                                        or (Iteration >= ItMax);

  if ( Iteration >= ItMax )
     and not( AttainedConvergence(NewResult, Deviation, Residual) ) then
                                                ErrorDiagnostic(2);

1:  {... BandCore exit label ...}

  LastIteration := true;

AuxiliaryManipulations( LastIteration, Iteration,
                                NewResult, Deviation, Residual );


FinalResult := NewResult
```

```
end;  { BandCore }


procedure Exit;

{...    Purpose:  Exit prints an error message and terminates the
                  program execution.
                                                                ...}
begin

        writeln;
        write('PROGRAM EXECUTION ABORTED');
        writeln;
          goto 1   {... break and go to BandShell exit label below }

end;  { Exit }



begin  { body of BandShell }

        Iteration := 0;
        ClockInitial := SystemClock;
        MarkingEntryLocations := false;

        SetParameters;
        CheckParameters;
        SetMeshSize;
        SetXDist;


        SetSkipMatrix;


        BandCore( n, jMax, ItMax, FactorIncrement, AbsoluteIncrement,
                        Expression, AttainedConvergence, AuxiliaryManipulations,
                        Guess, FinalResult, Deviation, Residual, Iteration );


1:  {... BandShell exit label ...}

          CPUTime := SystemClock - ClockInitial

end;  { BandShell }
```

Source Code for Procedure BandAid (File: BandAid.pas)

```
procedure BandAid
        ( n, jMax, ItMax : integer;
          var It, Time : integer;
          xMax, Tolerance : RealNumber;

          function AIDEquation
                        ( i, j : integer;
                          x, h : RealNumber;
                          var NewResult : ValueArray;
                          function cInterp
                                ( k : integer;
                                  x : RealNumber;
                                  var Result : ValueArray ) : RealNumber;
                          function AIDdnFdxn
                                ( n : integer;
                                  function F(x:RealNumber):RealNumber;
                                  x : RealNumber ) : RealNumber ) : RealNumber;

          Guess : ValueArray;
          var FinalResult : ValueArray;
          var Residual : ValueArray
                                                                    );

const     ImageFirstPoint = false;
        ImageLastPoint = false;
        FactorIncrement = 1e-6;
        AbsoluteIncrement = 1e-6;
        ReduceTimeOption = true;
          xMin = 0;


var       Deviation : ValueArray;




function Equation( i, j : integer;
                     x, h : RealNumber;
                     var NewResult : ValueArray;
                   function cInterp( i : integer;
                                     x : RealNumber;
                                     var Result : ValueArray ) : RealNumber;
                   function dnFdxn( n : integer;
                                    function F( x : RealNumber ) : RealNumber;
                                    x : RealNumber;
                                    Approx : DiffApprox ) : RealNumber
                                                                  ) : RealNumber;



          function Difference : DiffApprox;
          begin
                   if ( j = 1 ) then
                            Difference := For3PtDiff

                   else if ( j = jMax ) then
                            Difference := Back3PtDiff
```

```
                else      Difference := CenDiff
        end;

        function AIDdnFdxn( n : integer;
                            function F(x:RealNumber):RealNumber;
                            x : RealNumber ) : RealNumber;
        begin
                AIDdnFdxn := dnFdxn(n,F,x,Difference)
        end;


begin  { body of Equation }

        Equation := AIDEquation( i, j, x, h, NewResult, cInterp, AIDdnFdxn )

end;  { Equation }



function Converged( function x( j : integer ) : RealNumber;
                    h : RealNumber;
                    var NewResult, Deviation : ValueArray;
                    var Residual : ValueArray;
                  function cInterp( i : integer;
                                    x : RealNumber;
                                    var Result : ValueArray ) : RealNumber;
                  function dnFdxn( n : integer;
                                   function F( x : RealNumber ) : RealNumber;
                                   x : RealNumber;
                                   Approx : DiffApprox ) : RealNumber
                                                                ) : boolean;

var     k, j : integer;
        absDeviation, absValue : RealNumber;


begin

        Converged := true;

        for j := 1 to jMax do
                for k := 1 to n do

                begin
                        absDeviation := abs( Deviation[k,j] );
                           absValue := abs( NewResult[k,j] );

                           if ( absValue <> 0 ) then
                           if ( absDeviation > Tolerance*absValue ) then
                                        Converged := false

                end

end;  { Converged }
```

```
procedure NonBandCalcs( LastIteration : boolean;
                        Iteration, CPUTime : integer;
                            function x( j : integer ) : RealNumber;
                            h : RealNumber;
                        var NewResult, Deviation : ValueArray;
                            var Residual : ValueArray;
                        function cInterp( i : integer;
                                          x : RealNumber;
                                          var Result : ValueArray ) : RealNumber;
                        function dnFdxn( n : integer;
                                         function F( x : RealNumber ) : RealNumber;
                                         x : RealNumber;
                                         Approx : DiffApprox ) : RealNumber );

begin

        if LastIteration then
        begin
                It := Iteration;
                Time := CPUTime
        end

end; { NonBandCalcs }
```

```
begin { body of BandAid }

        BandShell( n, jMax, ItMax,
                xMin, xMax, FactorIncrement, AbsoluteIncrement,
                ImageFirstPoint, ImageLastPoint, ReduceTimeOption,
                Guess, FinalResult, Deviation, Residual,
                Equation, Converged, NonBandCalcs )

end; { BandAid }
```

Source Code for IOPkg, Procedures for Input/Output (File: IOPkg.pas)

```
{...

        IOPkg:  A collection of useful formatting procedures for
                input to and output from BandAid calling routines.

                                                                    ...}


procedure FIND( c : char );   {... find next occurence of a character c }

var     ch : char;

begin
        repeat read(ch) until ( EOF or (ch = c) )
end;


procedure RR( var r : RealNumber );          {... read a RealNumber r }
begin
        FIND('='); read(r)
end;

procedure RI( var i : integer );             {... read an integer i }
begin
        FIND('='); read(i)
end;



procedure StringRead( var s : string );   {... read a string s,
                                                (generic version,
                                                  used below) }

var     c : char;
        i : integer;
begin

        repeat read(c) until ( c = '<' );

        i := 0;

        repeat

                read(c);

                if ( c <> '>' ) then
                begin
                        i := i + 1;
                        s.chars[i] := c
                end

        until ( c = '>' );

        s.length := i

end;  { StringRead }


procedure RS( var s : string );              {... read a string s }
begin
```

```
        FIND('=');
        StringRead(s)

end;  { RS }


procedure RRowRead( var v : Vector );                {... read a vector v,
                                                          or a row of a matrix
                                                          (generic version,
                                                            used below) }

var     c : char;
        i : integer;

begin

        i := 0;

        repeat  read(c) until ( c = '<' );

        repeat
                i := i + 1;
                read(v[i]);

                repeat  read(c) until ( c = ',' ) or ( c = '>' )

        until ( c = '>' ) or EOF

end;  { RRowRead }



procedure IRowRead( var v : IVector );               {... read an integer-vector v,
                                                          or a row of a matrix
                                                          (generic version,
                                                            used below) }

var     c : char;
        i : integer;

begin

        i := 0;

        repeat  read(c) until ( c = '<' );

        repeat
                i := i + 1;
                read(v[i]);

                repeat  read(c) until ( c = ',' ) or ( c = '>' )

        until ( c = '>' ) or EOF

end;  { IRowRead }



procedure RV( var v : Vector );                      {... read a vector v }
```

```
begin

        FIND('=');
        RRowRead(v)

end; { RV }



procedure RIV( var v : IVector );          {... read an integer-vector v }
begin

        FIND('=');
        IRowRead(v)

end;  { RIV }



procedure RM( var m : Matrix );            {... read a matrix m }

var     c : char;
        j : integer;

begin

        FIND('=');

        j := 0;

        repeat  read(c)  until ( c = '<' );

        repeat

                j := j + 1;
                RRowRead(m[j]);

                repeat  read(c)  until  ( c = ',' ) or ( c = '>' )

        until ( c = '>' ) or EOF

end;  { RM }



procedure RIM( var m : IMatrix );                {... read an integer-matrix m }

var     c : char;
        j : integer;

begin

        FIND('=');

        j := 0;

        repeat  read(c)  until  ( c = '<' );

        repeat
```

```
                j := j + 1;
                IRowRead(m[j]);

                repeat  read(c)  until  ( c = ',' ) or ( c = '>' )

        until ( c = '>' ) or EOF

end;  { RIM }


procedure TB( n : integer );                    {... print n blank spaces (tab) }

begin
        write(' ':n)

end;  { TB }


procedure LF( n : integer );                    {... print n blank lines (line feeds) }

var     lines : integer;

begin
        for lines := 1 to n do
                writeln

end;  { LF }


procedure WR( r : RealNumber; e1, e2 : integer );  {... write a RealNumber r,
                                                        e1 and e2 are field
                                                        lengths  ...}

begin

        if ( abs(r) = 0 ) then
                write(r:e1:e2,' ':4)                    {... decimal notation ...}

        else if ( abs(r) >= 0.001 ) and ( abs(r) < 0.01 ) then
                write(r:(e1+2):(e2+2),' ':2)

        else if ( abs(r) >= 0.01 ) and ( abs(r) < 0.1 ) then
                write(r:(e1+1):(e2+1),' ':3)

        else if ( abs(r) >= 0.1 ) and ( abs(r) < 1 ) then
                write(r:e1:e2,' ':4)

        else if ( abs(r) >= 1 ) and ( abs(r) < 10 ) then
                write(r:(e1-1):(e2-1),' ':5)

        else if ( abs(r) >= 10 ) and ( abs(r) < 100 ) then
                write(r:(e1-2):(e2-2),' ':6)

        else if ( abs(r) >= 100 ) and ( abs(r) < 1000 ) then
                write(r:(e1-3):(e2-3),' ':7)
```

```
        else
        if ( abs(r) < 0.001 ) or ( abs(r) >= 1000 ) then
        begin
                if ( (e1-e2) >= 3) then
                        write(' ':(e1-e2-3));
                write(r:(e2+6),' ':1)                   {... scientific notation ...}
        end

        else
                write(r:e1:e2,' ':4);                   {... decimal notation ...}

end; { WR }


procedure WI( i : integer; e1 : integer );              {... write an integer i }
begin
        write(i:e1)
end; { WI }



procedure WS( s : string );             {... write a string s }

var     i : integer;

begin

        for i := 1 to s.length do write(s.chars[i])

end; { WS }



procedure WV( v : Vector;
              e1, e2 : integer;
              l : integer );                    {... write a vector v }

var     i : integer;

begin

        for i := 1 to l do
        begin
                TB(1); WR(v[i],e1,e2)
        end;

        LF(1);

end; { WV }



procedure WIV( v : IVector;
               e1 : integer;
               l : integer );                   {... write an integer-vector v }

var     i : integer;

begin
```

```
        for i := 1 to l do
        begin
                TB(1); WI(v[i],e1)
        end;

        LF(1);

end;  { WIV }


procedure WM( m : Matrix;
              e1, e2 : integer;
              l1, l2 : integer );          {... write a matrix m }

var     i1, i2 : integer;

begin

        for i1 := 1 to l1 do
        begin
                LF(1); TB(15);

                for i2 := 1 to l2 do
                begin
                        TB(2); WR(m[i1,i2],e1,e2)
                end
        end;

        LF(1)

end;  { WM }


procedure WIM( m : IMatrix;
               e1 : integer;
               l1, l2 : integer );         {... write an integer-matrix m }

var     i1, i2 : integer;

begin

        for i1 := 1 to l1 do
        begin
                LF(1); TB(15);

                for i2 := 1 to l2 do
                begin
                        TB(2); WI(m[i1,i2],e1)
                end
        end;

        LF(1)

end;  { WIM }
```

Source Code for ListPrint (File: ListPrint.pas)

```
procedure ListPrint( Result : ValueArray;
                     function z( x : RealNumber ) : RealNumber;
                     n, jMax : integer;
                     ImageFirstPoint, ImageLastPoint : boolean;
                     xMin, xMax : RealNumber );

var      x, h : RealNumber;
         k, j : integer;

begin

         if ImageFirstPoint and ImageLastPoint then
                  h := ( xMax - xMin )/( jMax - 3 )

         else if ImageFirstPoint and not(ImageLastPoint) then
                  h := ( xMax - xMin )/( jMax - 2 )

         else if not(ImageFirstPoint) and ImageLastPoint then
                  h := ( xMax - xMin )/( jMax - 2 )

         else if not(ImageFirstPoint) and not(ImageLastPoint) then
                  h := ( xMax - xMin )/( jMax - 1 );

         LF(1);

         for j := 1 to jMax do
         begin
                  if ImageFirstPoint then
                           x := xMin + h*(j - 2)
                  else     x := xMin + h*(j - 1);

                  TB(2); WI(j,4); TB(3); WR(z(x),10,5);

                  for k := 1 to n do
                  begin
                           TB(3); WR(Result[k,j],12,7)
                  end;

                  LF(1)

         end

end; { ListPrint }
```

Source Code for CatIO (File: CatIO.pas)

```
[INHERIT('AidMod.pen'),ENVIRONMENT('CatIO.pen')]

module CatIO ( input, output );


procedure ReadAndPrintParameters
                ( var DA, DB, cAo, cBo, v, L, kf, kb : RealNumber;
                  var jMax, ItMax : integer );

begin
        RR(DA); RR(DB); RR(cAo); RR(cBo); RR(v);
                RR(L); RR(kf); RR(kb);

                        RI(jMax);
                        RI(ItMax);

        LF(2); TB(25); write('CatReac Program');

        LF(2); TB(10); write('Parameters');

        LF(2); TB(15); write('DA = '); WR(DA,10,5); write(' cm2/sec');
        LF(1); TB(15); write('DB = '); WR(DB,10,5); write(' cm2/sec');
        LF(1); TB(15); write('cAo = '); WR(cAo,10,5); write(' mol/cm3');
        LF(1); TB(15); write('cBo = '); WR(cBo,10,5); write(' mol/cm3');
                        TB(10); write('jMax = '); WI(jMax,1);
        LF(1); TB(15); write('v = '); WR(v,10,5); write(' cm/sec');
                        TB(13); write('ItMax = '); WI(ItMax,1);
        LF(1); TB(15); write('L = '); WR(L,10,5); write(' cm');
        LF(1); TB(15); write('kf = '); WR(kf,10,5); write(' cm3/mol-sec');
        LF(1); TB(15); write('kb = '); WR(kb,10,5); write(' 1/sec');


end; { ReadAndPrintParameters }



procedure PrintOut( Result : ValueArray;
                    jMax, It, CPUTime : integer;
                    L : RealNumber );

var     j : integer;
        x, xMin : RealNumber;
        ImageFirstPoint, ImageLastPoint : boolean;

function z( x : RealNumber ) : RealNumber;
begin   z := x          end;


begin

        ImageFirstPoint := false;
        ImageLastPoint := false;


        xMin := 0;

        LF(3); TB(25); write('Profile Listing');
        LF(2); TB(5); write('Number of iterations = '); WI(It,1);
        LF(1); TB(5); write('Execution time = ');
                        WI(CPUTime,1); write(' milli-seconds');
```

```
            LF(3);
            TB(5); write('j'); TB(9); write('x');
            TB(16); write('cA'); TB(17); write('cB');
            LF(1);


            ListPrint( Result, z, 2, jMax, ImageFirstPoint, ImageLastPoint,
                                                                xMin, L     );

            LF(2); TB(25); write('End of Listing');

            LF(3)

end; { PrintOut }


end. { CatIO }
```

Sample Data File for CatReac (File: CatReac.dat)

```
DA = 1.0e-6
DB = 1.0e-6
cAo = 2e-1
cBo = 0.5e-1
v = 0.005
L = 10.0
kf = 1e-1
kb = 1e-3
jMax = 26
ItMax = 12
```

Source Code for Flow10 (File: Flow10.pas)

```
[INHERIT('AidMod.pen'),ENVIRONMENT('FlowIO.pen')]

module FlowIO ( input, output );



procedure ReadAndPrintParameters
                ( var P1, P2, P3, P4, DP, AlphaAR, AlphaCR,
                        AlphaAS, AlphaCS, AlphaL, IStar : RealNumber;
                  var jMax, ItMax : integer );

begin
        RR(P1); RR(P2); RR(P3); RR(P4); RR(DP);
        RR(AlphaAR); RR(AlphaCR);
        RR(AlphaAS); RR(AlphaCS); RR(AlphaL); RR(IStar);

                        RI(jMax);
                        RI(ItMax);

        LF(2); TB(25); write('FlowThru Program');

        LF(2); TB(10); write('Parameters');

        LF(2); TB(15); write('P1 = '); WR(P1,10,5);
           TB(10); write('AlphaAR =  ',AlphaAR:5:2);
        LF(1); TB(15); write('P2 = '); WR(P2,10,5);
           TB(10); write('AlphaCR =  ',AlphaCR:5:2);
        LF(1); TB(15); write('P3 = '); WR(P3,10,5);
           TB(10); write('AlphaAS =  ',AlphaAS:5:2);
        LF(1); TB(15); write('P4 = '); WR(P4,10,5);
           TB(10); write('AlphaCS =  ',AlphaCS:5:2);
        LF(1); TB(15); write('DP = '); WR(DP,10,5);
           TB(10); write('AlphaL = '); WR(AlphaL,9,4);
        LF(1); TB(15); write('I-Star = '); WR(IStar,10,5);

        LF(2); TB(15); write('jMax = '); WI(jMax,1);
        LF(1); TB(15); write('ItMax = '); WI(ItMax,1)


end; { ReadAndPrintParameters }



procedure PrintOut( Result : ValueArray;
                    jMax, It, CPUTime : integer;
                    AlphaL : RealNumber );

var     j : integer;
        y, yMin : RealNumber;
        ImageFirstPoint, ImageLastPoint : boolean;

function z( y : RealNumber ) : RealNumber;
begin   z := y          end;


begin


        ImageFirstPoint := false;
        ImageLastPoint := false;
```

```
        yMin := 0;


        LF(3); TB(25); write('Profile Listing');
        LF(2); TB(5); write('Number of iterations = '); WI(It,1);
        LF(1); TB(5); write('Execution time = ');
                        WI(CPUTime,1); write(' milli-seconds');
        LF(3);
        TB(5); write('j'); TB(9); write('x');
        TB(16); write('T '); TB(17); write('E ');
        LF(1);


        ListPrint( Result, z, 2, jMax, ImageFirstPoint, ImageLastPoint,
                                                        yMin, AlphaL      );

        LF(2); TB(25); write('End of Listing');

        LF(3)
end; { PrintOut }


end. { FlowIO }
```

Sample Data File for FlowThru (File: FlowThru.dat)

```
P1 = 104.9e-9
P2 = -3.254
P3 = 1.247e-5
P4 = 5.863e-9
DP = 12.17e-2
AlphaAR = 1.5
AlphaCR = 0.5
AlphaAS = 0.5
AlphaCS = 0.5
AlphaL = 8.663
IStar = 0.95

jMax = 26
ItMax = 15
```

Source Code for KarmanIO (File: KarmanIO.pas)

```
[INHERIT('AidMod.pen'),ENVIRONMENT('KarmanIO.pen')]

module KarmanIO ( input, output );


procedure ReadAndPrintParameters
                  ( var jMax, ItMax : integer );

begin

                              RI(jMax);
                              RI(ItMax);

          LF(2); TB(25); write('vonKarman Program');

          LF(2); TB(10); write('Parameters');

          LF(2); TB(15); write('jMax = '); WI(jMax,1);
          LF(1); TB(15); write('ItMax = '); WI(ItMax,1);


end; { ReadAndPrintParameters }



procedure PrintOut( Result : ValueArray;
                    jMax, It, CPUTime : integer;
                    L : RealNumber );

var       j : integer;
          x, xMin : RealNumber;
          ImageFirstPoint, ImageLastPoint : boolean;

function z( x : RealNumber ) : RealNumber;
begin     z := x              end;

begin

          ImageFirstPoint := false;
          ImageLastPoint := false;


          xMin := 0;

          LF(3); TB(25); write('Profile Listing');
          LF(2); TB(5); write('Number of iterations = '); WI(It,1);
          LF(1); TB(5); write('Execution time = ');
                              WI(CPUTime,1); write(' milli-seconds');
          LF(3);
          TB(5); write('j'); TB(9); write('z');
          TB(16); write('F '); TB(17); write('G ');
          TB(17); write('H '); TB(17); write('P ');
          LF(1);


          ListPrint( Result, z, 4, jMax, ImageFirstPoint, ImageLastPoint,
                                                          xMin, L      );

          LF(2); TB(25); write('End of Listing');
```

```
        LF(3)
end;  { PrintOut }

end.  { Karman10 }
```

Sample Data File for vonKarman (File: vonKarman.dat)

```
jMax = 101
ItMax = 15
```

Source Code for LSVIO (File: LSVIO.pas)

```
[INHERIT('AidMod.pen'),ENVIRONMENT('LSVIO.pen')]

module LSVIO ( input, output );


procedure ReadAndPrintParameters
                ( var m, jMax, ItMax : integer;
                  var D, z, s, co : Vector;
                  var Omega, nu, ioref, Uo, beta,
                        n, b, dU, UMax, xMax : RealNumber );

begin
        RI(m); RI(jMax); RI(ItMax);

        RV(D); RV(z); RV(s); RV(co);

        RR(Omega); RR(nu); RR(ioref); RR(Uo); RR(beta);
                RR(n); RR(b); RR(dU); RR(UMax); RR(xMax);

        LF(2); TB(25); write('LSV Program');

        LF(2); write('Parameters:');

          LF(2); TB(10); write('Number of Species = '); WI(m,1);
          LF(1); TB(10); write('Number of Meshpoints = '); WI(jMax,1);
          LF(1); TB(10); write('Maximum Iterations = '); WI(ItMax,1);

          LF(3); write('Diffusion coefficients (D[k], cm2/s):'); WV(D,10,5,m);

        LF(2); write('Valences (z[k]):'); WV(z,10,5,m);

        LF(2); write('Stoichiometric Coefficients (s[k]):'); WV(s,5,2,m);

        LF(2); write('Bulk Concentrations (co[k], mol/cc):'); WV(co,10,5,m);

        LF(4); TB(10); write('Omega = '); WR(Omega,10,5); write(' rad/s');
        LF(1); TB(10); write('nu = '); WR(nu,10,5); write(' cm2/s');
        LF(1); TB(10); write('ioref = '); WR(ioref,10,5); write(' A/cm2');
        LF(1); TB(10); write('Uo = '); WR(Uo,10,5); write(' V');
        LF(1); TB(10); write('beta = '); WR(beta,5,2);
        LF(1); TB(10); write('n = '); WR(n,5,2);
        LF(1); TB(10); write('b = '); WR(b,10,5); write(' V/s');
        LF(1); TB(10); write('dU = '); WR(dU,10,5); write(' V');
        LF(1); TB(10); write('UMax = '); WR(UMax,10,5); write(' V');
          LF(1); TB(10); write('xMax = '); WR(xMax,5,2);

        LF(5)

end; { ReadAndPrintParameters }



procedure PrintOut( Result : ValueArray;
                    L : RealNumber;
                    nSpcs, jMax, It, CPUTime : integer );

const   ImageFirstPoint = false;
        ImageLastPoint = false;
        yMin = 0;
```

```
var       k : integer;

function y( x : RealNumber ) : RealNumber;
begin     y := x              end;


begin

          LF(3); TB(25); write('Profile Listing');
          LF(2); TB(5); write('Number of iterations = '); WI(It,1);
          LF(1); TB(5); write('Execution time = '); WI(CPUTime,1);
                              write(' milli-seconds'); LF(1);

          LF(2); TB(5); write('j');  TB(9); write('y');

          for k := 1 to nSpcs do
          begin
                    TB(17); write('c',k:1)
          end;

          TB(17); write('V');

          LF(1);


          ListPrint( Result, y, nSpcs + 1, jMax,
                              ImageFirstPoint, ImageLastPoint, yMin, L )
end;


end  { LSVIO }
```

Sample Data File for LSV (File: LSV.dat)

```
m = 3
jMax = 21
ItMax = 12

D = < 1e-6, 1e-6, 1e-6 >
z = < +2, +1, -2 >    ( Cu++, SO4--, H+ )
s = < -1, 0, 0 >
co = < 0.1e-3,  1.8e-3,  1e-3 >

Omega = 44 rad/sec
nu = 1e-2 cm2/sec
ioref = 1e-3 A/cm2
Uo = 0 V
beta = 0.5
n = 1
b = -0.1 V/s
dU = 0.01 V
UMax = 0.02 V
xMax = 1.6
```

Source Code for GraetzIO (File: GraetzIO.pas)

```
[INHERIT('AidMod.pen'),ENVIRONMENT('GraetzIO.pen')]

module GraetzIO ( input, output );



procedure ReadAndPrintParameters
                    ( var m, jMax, ItMax : integer );

begin
                              RI(m);
                              RI(jMax);
                              RI(ItMax);

          LF(2); TB(25); write('Graetz Program');

          LF(2); TB(10); write('Parameters');

          LF(2); TB(15); write('m (eigenvalue number) = '); WI(m,1);
          LF(2); TB(15); write('jMax = '); WI(jMax,1);
          LF(1); TB(15); write('ItMax = '); WI(ItMax,1);


end; { ReadAndPrintParameters }



procedure PrintOut( Result : ValueArray;
                    jMax, It, CPUTime : integer;
                    xMin, xMax : RealNumber );

var       j : integer;
          x : RealNumber;
          ImageFirstPoint, ImageLastPoint : boolean;

function z( x : RealNumber ) : RealNumber;
begin     z := x              end;

begin

          ImageFirstPoint := false;
          ImageLastPoint := true;

          LF(3); TB(25); write('Profile Listing');
          LF(2); TB(5); write('Number of iterations = '); WI(It,1);
          LF(1); TB(5); write('Execution time = ');
                              WI(CPUTime,1); write(' milli-seconds');
          LF(3);
          TB(5); write('j'); TB(9); write('x');
          TB(16); write('R '); TB(17); write('I ');
          LF(1);


          ListPrint( Result, z, 2, jMax, ImageFirstPoint, ImageLastPoint,
                                                        xMin, xMax      );

          LF(2); TB(25); write('End of Listing');

          LF(3)
```

```
end;  { PrintOut }

end.  { GraetzIO }
```

Sample Data File for Graetz (File: Graetz.dat)

```
m = 3
jMax = 52
ItMax = 10
```

Appendix B-3. Source Listings for Programs Using the Superposition Principle to Simulate Cyclic Voltammograms

The programs shown in the following sections are the ones used for the simulation of cyclic voltammograms in Part 3. The format and organization of the programs is similar to that used in Part 2. The source code for SuperPose.pas and NewtRaph.pas and PrintOut.pas is included in module PoseMod.pas, which is compiled much the same as AidMod. The calling program, CycVolt is compiled and linked with PoseMod before execution. Procedure PrintOut uses the same input/output utilities (IOPkg.pas) that are shown in appendix B-2.

Source Code for PoseMod (File: PoseMod.pas)

```
[ENVIRONMENT('PoseMod.pen')]

module PoseMod( input, output );

const   StepMax = 1000;
        NMax = 5;
        NSpcsMax = NMax;

        SystemZero = 1e-30;

type    RealNumber = double;
        Vector = array [1..NMax] of RealNumber;
        IVector = array [1..NMax] of integer;
        Matrix = array [1..NMax] of Vector;
        IMatrix = array [1..NMax] of IVector;

        IterationStore = array [0..StepMax] of integer;
        ErrorStore = array [0..StepMax] of boolean;
        ResultStore = array [0..StepMax] of RealNumber;

        SurfaceValueArray = array [1..NSpcsMax] of ResultStore;



        string = record chars : array [1..100] of char;
                        length : integer            end;



        %include 'NewtRaph.pas'
        %include 'SuperPose.pas'
        %include 'IOPkg.pas'
        %include 'PrintOut.pas'

end  { PoseMod module }
```

Source Code for NewtonRaphson (File: NewtRaph.pas)

```
{...

        ================================================

        Procedure Title:  NewtonRaphson
        Written by:       Michael Matlosz

        Date:     April 17, 1984
        UpDated:  August 26, 1984 and December 15,1984

                  Copyright (C) 1985 by Michael Matlosz
                  All rights reserved.


        Purpose:        Determine the solution vector c[i] that satisfies
                  the system of equations F(i,c) = 0.  The technique employed
                  is a multi-dimensional Newton-Raphson method.  Derivatives
                  are determined numerically, and the coefficient matrix
                  (Jacobian) is inverted using the matrix inversion algorithm
                  of Newman (MATINV), which is reproduced here in Pascal as
                  the subprogram MatrixInversion.  The routine is intended
                  to be used with procedure Superpose to solve the multi-
                  component diffusion equations occurring in cyclic-voltammetry
                  problems.  (See module PoseMod for a listing of the
                  SuperPose procedure.)

        ================================================

                                                                    ...}


procedure NewtonRaphson( function FTrial( i : integer;
                                          c : Vector ) : RealNumber;
                         cGuess : Vector;
                         N, ItLim : integer;
                         Tolerance : RealNumber;
                         var     TotalIterations : integer;
                         var         cResult : Vector;
                         var         Err1, Err2 : boolean        );

label     2;

type      Matrix = array [1..NMax, 1..NMax] of RealNumber;

var     cNew, cOld, cDiff, dcdxOld : Vector;
        Iteration : integer;
          Determinant : RealNumber;

          F : array [1..NMax] of RealNumber;
          dFdc, dFdcInverse : Matrix;


          DetermIsZero : boolean;


          i, j : integer;
```

```
procedure MatrixInversion( M : Matrix;
                           var Minverse : Matrix;
                               N : integer          );

{... Purpose: Determine the inverse, Minverse, of the square (N by N)
              matrix M.

     Method: Gaussian Elimination using elementary row operations.
             The algorithm is adapted from Newman's subroutine MATINV.
             For each of the N row-elimination steps, the following
             four steps are repeated:

             Step 1: Determination of the pivot. Go through the
             rows of M, one at a time (skipping rows already used),
             in order to determine the location of the largest entry
             (in absolute value) in the row with the smallest ratio
             of second-largest entry to largest entry (i. e., the
             smallest ratio NextToMaxEntry/MaxEntry). Thus, the
             BestMaxEntry is the largest entry in the row with the
             BestRatio. This BestMaxEntry will become the Pivot.
             (This choice of pivot reduces roundoff error.)

             Step 2: Row interchange. If the BestMaxEntry (the choice
             for pivot) is not on the diagonal of M, then two rows
             are interchanged such that BestMaxEntry is on the diagonal.

             Step 3: Division by Pivot. BestMaxEntry becomes the
             Pivot, and each element of the row containing the Pivot
             is divided by Pivot. (The diagonal entry of this row
             of M is now unity.)

             Step 4: Elimination. All entries in the column containing the
             Pivot (except the Pivot itself) are eliminated by suitable
             row multiplications and subtractions.

     Variables global to the routine: from calling routine -- DetermIsZero,
                                                                      Size

                                                                      ...}

label     1;

var  Row, Column, PivotColumn, PivotRow, RowChoice : integer;
     NumberOfRowEliminations, ColWithMaxRowEntry : integer;

     UsedRow, UsedCol : array [1..NMax] of boolean;

     MaxRowEntry, NextToMaxEntry : RealNumber;
     PresentRatio, BestRatio : RealNumber;
     Multiplier, Pivot, SavedValue : RealNumber;


procedure Search(Row:integer);

{... Purpose: Search through a row of matrix M to find the largest
              entry of the row (MaxRowEntry) and the second-largest
              entry of the row (NextToMaxEntry). Also, indicate the
              column containing the MaxRowEntry, and activate the
              DetermIsZero flag if the row contains only zeros.

     Variables global to the routine: from MatrixInversion --
                                           M, N, ColWithMaxRowEntry,
```

```
                                        MaxRowEntry, NextToMaxEntry,
                                        UsedCol, DetermIsZero

                                                                ...}

var   Column : integer;

begin  {body of Search}

       MaxRowEntry := 0;  NextToMaxEntry := 0;

       for Column := 1 to N do
            if not UsedCol[Column] then

              if abs(M[Row,Column]) > MaxRowEntry then
                        begin
                             NextToMaxEntry := MaxRowEntry;
                             MaxRowEntry := abs(M[Row,Column]);
                             ColWithMaxRowEntry := Column
                        end

                   else if abs(M[Row,Column]) > NextToMaxEntry then

                             NextToMaxEntry:=abs(M[Row,Column]);


       if MaxRowEntry=0 then DetermIsZero := true

end;  {Search}



begin  {body of GaussElimination}

   {... Initializations ...}

        for Row := 1 to N do
                for Column := 1 to N do
                        if (Row = Column) then
                                Minverse[Row,Column] := 1
                        else    Minverse[Row,Column] := 0;

       DetermIsZero := false;

       for Row := 1 to N do UsedRow[Row]:=false;
       for Column := 1 to N do UsedCol[Column]:=false;


   {... Solve the equations ...}

       for NumberOfRowEliminations := 1 to N do

       begin  { row eliminations }


          {... Step 1:  Pivot Determination ...}

                BestRatio := 1.1;   {... setting BestRatio to 1.1 guarantees
                                         that the test "if PresentRatio < BestRatio"
```

```
                                        below will fail on the first pass
                                                                    ...}
    for Row := 1 to N do
    if not UsedRow[Row] then
    begin
            Search(Row);
            if DetermIsZero then goto 1;

            PresentRatio := NextToMaxEntry/MaxRowEntry;

            if PresentRatio <= BestRatio then
                begin
                    BestRatio := PresentRatio;

                    RowChoice := Row;
                    PivotColumn := ColWithMaxRowEntry
                end

    end;

    PivotRow := PivotColumn;
    UsedCol[PivotColumn] := true;


{... Step 2:  Row Interchange ...}

    if RowChoice <> PivotRow then
    begin
            for Column := 1 to N do
            begin
                    SavedValue := M[RowChoice,Column];
                    M[RowChoice,Column] := M[PivotRow,Column];
                    M[PivotRow,Column] := SavedValue
            end;

            for Column := 1 to N do
            begin
                    SavedValue := Minverse[RowChoice,Column];
                    Minverse[RowChoice,Column] := Minverse[PivotRow,Column];
                    Minverse[PivotRow,Column] := SavedValue
            end
    end;

    UsedRow[PivotRow]:=true;


{... Step 3:  Divide by Pivot ...}

    Pivot := M[PivotRow,PivotColumn];

    for Column:=1 to N do M[PivotRow,Column]:=
                                        M[PivotRow,Column]/Pivot;
    for Column:=1 to N do Minverse[PivotRow,Column]:=
                                        Minverse[PivotRow,Column]/Pivot;


{... Step 4:  Elimination ...}

    for Row := 1 to N do
    if Row <> PivotRow then
    begin
            Multiplier := M[Row,PivotColumn];
```

```
                    for Column := 1 to N do M[Row,Column] := M[Row,Column]
                                        - Multiplier*M[PivotRow,Column];
                    for Column := 1 to N do Minverse[Row,Column] := Minverse[Row,Column]
                                        - Multiplier*Minverse[PivotRow,Column]
                end;

        end;  { row eliminations }


1:      if DetermIsZero then
          begin
                  Err1 := true;
                  goto 2
          end

end;  { MatrixInversion }



function NumericalDeriv( c : Vector;
                         i, j : integer ) : RealNumber;

{...    Purpose:  Compute dFdc by numerical differentiation    ...}

const   Small = 1e-6;

var     epsilon : RealNumber;
          cIncremented : Vector;

begin  { body of NumericalDeriv }

          cIncremented := c;

        epsilon := abs(c[j])*Small;
        if (epsilon < SystemZero) then epsilon := small;

          cIncremented[j] := c[j] + epsilon;

          NumericalDeriv := ( FTrial(i,cIncremented)
                                        - FTrial(i,c) )/epsilon

end;  { NumericalDeriv }



function Converged( cOld, cNew : Vector ) : boolean;

{...        Purpose:  Determine if a solution has been found  ...}

var     j : integer;

begin
          Converged := true;

          for j := 1 to N do
              if ( abs(cOld[j] - cNew[j]) > Tolerance*cOld[j] ) then
```

```
                                                        Converged := false
end;  { Converged }


procedure MatrixVectorMult( M : Matrix;
                            V : Vector;
                            var MV : Vector;
                            N : integer        );

{...        Purpose:  Multiply a square (N by N) matrix M by a vector V of
                      length N to produce the vector MV.                    ...}

var         i, j : integer;
            Sum : RealNumber;

begin
            for i:= 1 to N do
            begin
                        Sum := 0;
                        for j := 1 to N do
                                    Sum := Sum + M[i,j]*V[j];
                        MV[i] := Sum
            end

end;  { MatrixVectorMult }


begin  { body of NewtonRaphson }


              Err1 := false;
              Err2 := false;

           Iteration := 0;
           cNew := cGuess;

           repeat
                   Iteration := Iteration + 1;
                   cOld := cNew;

                      for i := 1 to N do
                      begin
                              F[i] := FTrial(i,cOld);

                              for j := 1 to N do
                                      dFdc[i,j] := NumericalDeriv(cOld,i,j)
                      end;


                      MatrixInversion(dFdc, dFdcInverse, N);

                      MatrixVectorMult( dFdcInverse, F, cDiff, N );


                      for j := 1 to N do
                      begin
                              cNew[j] := cOld[j] - cDiff[j];
                              if (cNew[j] <= 0) then
```

```
                                        cNew[j] := 0.1*cOld[j]
                    end;

          until     Converged(cOld,cNew) or (Iteration >= ItLim);

2:          cResult := cNew;
          TotalIterations := Iteration;

          if (Iteration >= ItLim) then Err2 := true

end;  { NewtonRaphson }
```

Source Code for SuperPose (File: SuperPose.pas)

{...

==========

Procedure Title: SuperPose
Written by: Michael Matlosz

Date: April 17, 1984
UpDated: August 26, 1984 and December 15, 1984

Purpose: *SuperPose is a subprogram for the solution of the unsteady-state, multi-component diffusion equations (in stagnant electrolyte, i. e., no convection) at a planar electrode during cyclic voltammetry. The routine uses a multidimensional Newton-Raphson routine (found in proceudre NewtonRaphson of module NewtMod) to determine the appropriate change in concentration of each species at a each time step, such that the boundary conditions at the electrode surface (supplied by the calling program) are satisfied. The flux of each species to the electrode surface (at each time interval) is computed from the super-position theorem (Duhamel's integral), constructed from the solution to the unsteady-state diffusion equation in a semi-infinite stagnant medium (or a Nernst stagnant diffusion layer) resulting from a step change in surface concentration. The integral is evaluated by the method of Acrivos and Chambre.*

==========
...}

```
procedure SuperPose (   NSpcs : integer;
                        nStart : integer;
                        var nStop : integer;
                        D, kFilm : Vector;
                        tStart, tStop, dt : RealNumber;
                        var c, dcdx : SurfaceValueArray;
                        var t : ResultStore;
                        var Iterations : IterationStore;
                        var Err1, Err2 : ErrorStore;
                        var CPUTime : integer;
                        function SurfBC( i : integer;
                                         c, dcdx : Vector;
                                         t : RealNumber ) : RealNumber;
                        ItMax : integer;
                        Tolerance : RealNumber           );

const   pi = 3.14159;
```

```
var     Acoeff : SurfaceValueArray;
        cGuess, cResult, IntegralSoFar, del : Vector;
        i, n, ClockStart, ClockStop : integer;


procedure SetCoeffs;

{...    Purpose:  Calculate the various coefficients needed for the calculation
                  of the superposition integral.  These values are only functions
                  of the index (n - k), where n is the step and k is the summation
                  index.  Here, n is set to StepMax so that all possible
                  Acoeff[n-k] can be evaluated.

        Variables global to the routine:        from IodideCV -- Acoeff,
                                                           StepMax
                                                                        ...}

const   n = StepMax;

var     i, k : integer;

        function a( i : integer; t : RealNumber ) : RealNumber;

        const   mMax = 5;

        var     System : ( SemiInfinite, NernstLayer );
                Sum, Term, Dbetasqr : RealNumber;
                m : integer;

        begin

            if (kFilm[i] = 0) then
                        System := SemiInfinite

            else
            if ( t <= 4*sqr(del[i])/(pi*D[i]) ) then
                        System := SemiInfinite

            else        System := NernstLayer;


            case System of

                SemiInfinite:
                begin

                        a := -2*sqrt( t/(pi*D[i]) )


                end;  { SemiInfinite case }

                NernstLayer:
                begin
```

```
                    Sum := 0;

                    for m := 1 to mMax do
                    begin

                            Dbetasqr := D[i]*sqr( m*pi/del[i] );

                            Term := (2/Dbetasqr)*( exp(-Dbetasqr*t) );

                            Sum := Sum + Term
                    end;


                    a := -t/del[i] - Sum/del[i]

                end  { NernstLayer case }

            end  { System cases }

        end;  { a }



begin  { body of SetCoeffs }

    for k := 0 to (n - 1) do
        for i := 1 to NSpcs do

                Acoeff[i,n-k] := a(i,(n-k)*dt) - a(i,(n-k-1)*dt)

end;  { SetCoeffs }



function LeadingTerms( i, n : integer ) : RealNumber;

{...    Purpose:  For a given step n, compute the "leading terms" in the
                  summation representation of the superposition integral, i. e.,
                  the sum from k=0 to n-2 (since the total sum runs from
                  k=0 to n-1).  Because the weights for the present step are
                  as yet unknown, the complete integral cannot be calculated.
                  However, all terms involving weights from previous steps
                  can be calculated, and by computing this part of the
                  integral prior to the call to the Newton routine (which
                  involves iterations) the execution time of the program
                  can be reduced substantially.

                                                                          ...}

var     k : integer;
        Sum : RealNumber;

begin  { body of LeadingTerms }

        if ( n < 2 ) then
        begin

                Sum := 0
```

```
        end  { if }

        else
        begin

                Sum := 0;

                for k:= 0 to (n-2) do

                        Sum := Sum + ( (c[i,k+1] - c[i,k])/dt )*Acoeff[i,n-k]

        end;  { else }

      LeadingTerms := Sum

end;  { LeadingTerms }



function CompleteIntegral( i, n : integer;
                           cNew : RealNumber;
                           IntegralSoFar : Vector ) : RealNumber;

{...    Purpose:  From the IntegralSoFar (i. e., the LeadingTerms), compute
                  a complete superposition integral, by adding the last term
                  (i. e., the term for k = n - 1).

                                                                          ...}

var     k : integer;


begin { CompleteIntegral }

        k := n - 1;

        CompleteIntegral := IntegralSoFar[i] +
                                ( (cNew - c[i,k])/dt )*Acoeff[i,n-k]

end;  { CompleteIntegral }



function BCTrial( eqn : integer; cTrial : Vector ) : RealNumber;

var     i : integer;
        dcdxTrial : Vector;

begin  { body of BCTrial }

        for i := 1 to NSpcs do
        begin
                dcdxTrial[i] := CompleteIntegral(i, n, cTrial[i], IntegralSoFar )
        end;

        BCTrial := SurfBC(eqn,cTrial,dcdxTrial,t[n])

end;  { BCTrial }
```

```
begin  { body of SuperPose }

        ClockStart := Clock;

        n := nStart;
        t[nStart] := tStart;
        Iterations[nStart] := 0;

        for i := 1 to NSpcs do
                if ( kFilm[i] <> 0 ) then
                        del[i] := D[i]/kFilm[i];

        SetCoeffs;

        repeat

            n := n + 1;
            t[n] := t[n-1] + dt;

            for i := 1 to NSpcs do
                IntegralSoFar[i] := LeadingTerms(i,n);

            for i := 1 to NSpcs do
                        cGuess[i] := c[i,n-1];

            NewtonRaphson( BCTrial, cGuess, NSpcs, ItMax, Tolerance,
                        Iterations[n], cResult, Err1[n], Err2[n] );

            for i := 1 to NSpcs do c[i,n] := cResult[i];

          for i := 1 to NSpcs do
                dcdx[i,n] := CompleteIntegral( i, n, c[i,n], IntegralSoFar )

        until Err1[n] or (t[n] >= (tStop - dt/2)) or (n >= StepMax);

        nStop := n;

        ClockStop := Clock;

        CPUTime := ClockStop - ClockStart
end;  { SuperPose }
```

Source Code for PrintOut (File: PrintOut.pas)

```
procedure PrintSweep( InitialStep, FinalStep : integer;
                      V, I : ResultStore;
                      IterationValue : IterationStore;
                      Err1, Err2 : ErrorStore;
                      ClockTime : integer );

var     Step : integer;

begin
        LF(5); TB(25); write('Final Results: ');

        if ( V[FinalStep] > V[InitialStep] ) then
                write('Anodic-direction Sweep')
        else    write('Cathodic-direction Sweep');

        LF(3);

        TB(4); write('Step');
        TB(8); write('V (Volts)');
        TB(14); write('I (Amps)');
        TB(11); write('Iterations');

        LF(1);

        for Step := InitialStep to FinalStep do
        begin
            if ( Err1[Step] ) then
            begin
                LF(2); TB(20);
                write('WARNING: Zero determinant found');
                LF(1); TB(20);
                write('                in MatrixInversion routine.');
                LF(1); TB(20);
                write('PROGRAM EXECUTION INTERRUPTED.');
                LF(2)
            end

            else
            begin
                LF(1);
                TB(3); WI(Step,4);
                TB(6); WR(V[Step],11,6);
                TB(8); WR(I[Step],11,6);
                TB(10); WI(IterationValue[Step],2);
                if ( Err2[Step] ) and ( Step <> InitialStep ) then
                                                        write(' (Limit) ')
            end

        end;

        LF(3);

        TB(25); write(' Sweep End  (Execution Time = ');
                        WI(ClockTime,1); write(' milli-seconds)');

        LF(5)

end;  { PrintSweep }
```

```
procedure PrintTitle;

var     DateString, TimeString : packed array [1..11] of char;

begin

        Date(DateString);  Time(TimeString);

        LF(5); TB(25);
        write(' Cyclic Voltammogram Program');

        LF(2); TB(25);
        write(' Written by:  Michael Matlosz');
        LF(1); TB(25);
        write(' Last Update: February 19, 1985');

        LF(3); TB(40);
        write(' Program begun at ',TimeString);
        LF(1); TB(40);
        write('                  on ',DateString);

        LF(5)

end; { PrintTitle }
```

Source Code for CycVolt (File: CycVolt.pas)

```
[INHERIT('PoseMod.pen')]

program CycVolt( input, output );

const   SpcsMax = 3;
        RxnMax = 3;

        R = 8.314;
        F = 96487;

        pi = 3.141592654;



var     C, dCdx : SurfaceValueArray;

        Co, ioRef, Utheta, alphaA, alphaC, n : Vector;
        Cref, s, p, q : Matrix;

        nSpcs, nRxns, MaximumIterations, TotalSweeps : integer;
        Ure, Tolerance : RealNumber;
        t, I, V : ResultStore;

        Iterations : IterationStore;
        Err1, Err2 : ErrorStore;

        tStart, tReversal, tRange, dt : RealNumber;

        StartStep, ReversalStep, CPUTime : integer;

        Sweep : integer;


        Vstart, Vreversal, dV : RealNumber;

        Uref : Vector;

        D, del, kFilm, z : Vector;


        kappa, radius,
                b, A, RotationSpeed, omega, rhoZero, Temp, nu : RealNumber;

        k : integer;


procedure ReadAndPrintParameters;

begin
        RI(nSpcs);
        RI(nRxns);
        RI(MaximumIterations);
        RR(Tolerance);

        RV(D);
        RV(z);
        RR(Ure);

        RR(Temp);
        RR(rhoZero);
```

```
RR(nu);


RV(Utheta);
RV(alphaA);
RV(alphaC);
RV(n);
RV(ioRef);
RM(s);
RM(Cref);

RV(Co);

RR(Vstart);
RR(Vreversal);
RR(dV);

RR(b);
RR(radius);
A := pi*sqr(radius);

RR(kappa);

RR(RotationSpeed);

RI(TotalSweeps);

LF(2);
TB(20); write('Parameters:');
LF(3);
TB(20); write('------ Input Data ------');

LF(2);
TB(10); write('Number of species (nSpcs) = '); WI(nSpcs,1);
LF(1);
TB(10); write('Number of reactions (nRxns) = '); WI(nRxns,1);
LF(2);
TB(10); write('MaximumIterations = '); WI(MaximumIterations,1);
LF(1);
TB(10); write('Tolerance = '); WR(Tolerance,10,5);
LF(2);
TB(10); write('D[k] (cm2/s) = '); WV(D,10,5,nSpcs);
LF(2);
TB(10); write('z[k] = '); WV(z,4,2,nSpcs);
LF(2);
TB(10); write('Ure (V) = '); WR(Ure,10,5);
LF(2);
TB(10); write('T (K) = '); WR(Temp,10,5);
LF(1);
TB(10); write('rhoZero (kg/cm3) = '); WR(rhoZero,10,5);
LF(1);
TB(10); write('nu (cm2/s) = '); WR(nu,10,5);
LF(2);
TB(10); write('Utheta[j] (V) = '); WV(Utheta,10,5,nRxns);
LF(1);
TB(10); write('alphaA[j] = '); WV(alphaA,4,2,nRxns);
LF(1);
TB(10); write('alphaC[j] = '); WV(alphaC,4,2,nRxns);
LF(1);
TB(10); write('ioRef[j] (A/cm2) = '); WV(ioRef,10,5,nRxns);
LF(2);
TB(10); write('s[k,j] = '); WM(s,10,5,nSpcs,nRxns);
LF(2);
```

```
        TB(10); write('Cref[k,j] = '); WM(Cref,10,5,nSpcs,nRxns);
        LF(3);
        TB(10); write('Co[k] = '); WV(Co,10,5,nSpcs);
        LF(1);
        TB(10); write('Vstart (V) = '); WR(Vstart,10,5);
        LF(1);
        TB(10); write('Vreversal (V) = '); WR(Vreversal,10,5);
        LF(1);
        TB(10); write('dV (V) = '); WR(dV,10,5);
        LF(1);
        TB(10); write('b (V/s) = '); WR(b,10,5);
        LF(1);
        TB(10); write('radius (cm) = '); WR(radius,10,5);
        LF(1);
        TB(10); write('A (cm2) = '); WR(A,10,5);
        LF(1);
        TB(10); write('kappa (mho/cm) = '); WR(kappa,10,5);
        LF(1);
        TB(10); write('RotationSpeed (rpm) = '); WR(RotationSpeed,10,5);
        LF(2);
        TB(10); write('TotalSweeps (forward and reverse) = ');
                                                    WI(TotalSweeps,1)

end; { ReadAndPrintParameters }



procedure SetParameters;

var     k, j : integer;

begin
        tStart := 0;

        if ( b <> 0 ) then
        begin
                dt := abs(dV/b);
                tReversal := abs( (Vreversal - Vstart)/b )
        end;

        tRange := tReversal - tStart;

        if ( Vreversal > Vstart ) then
                                b := abs(b)       {... anodic sweep first }
        else                    b := -abs(b);     {... cathodic sweep first }



        for j := 1 to nRxns do
        begin
            Uref[j] := Utheta[j] - Ure;

            for k := 1 to nSpcs do
                Uref[j] := Uref[j]
                        - ( s[k,j]*R*Temp/(n[j]*F) )*ln( Cref[k,j]/rhoZero )
        end;


        for j := 1 to nRxns do
                for k := 1 to nSpcs do
```

```
                if ( s[k,j] > 0 ) then
                begin
                        p[k,j] := s[k,j];
                        q[k,j] := 0
                end

                else if ( s[k,j] < 0 ) then
                begin
                        q[k,j] := -s[k,j];
                        p[k,j] := 0
                end

                else
                begin
                        p[k,j] := 0;
                        q[k,j] := 0
                end;


omega := RotationSpeed*(2*pi)/60;


for k := 1 to nSpcs do
begin

    if (RotationSpeed = 0) then kFilm[k] := 0
    else
    begin

        del[k] := 1.61*( D[k]**(1/3) )*( omega**(-1/2) )*( nu**(1/6) );

        kFilm[k] := D[k]/del[k]
    end

end;


LF(3);
TB(20); write('------ Derived Quantities ------');
LF(2);
TB(10); write('dt (s) = '); WR(dt,10,5);
LF(2);
TB(10); write('Uref[j] (V) = '); WV(Uref,10,5,nRxns);
LF(2);
TB(10); write('p[i,j] = '); WM(p,10,5,nSpcs,nRxns);
LF(2);
TB(10); write('q[i,j] = '); WM(q,10,5,nSpcs,nRxns);
LF(2);
TB(10); write('omega (rad/s) = '); WR(omega,10,5);
LF(1);
TB(10); write('del[i] (cm) = ');

         if ( RotationSpeed = 0 ) then
                write('undefined')
         else   WV(del,10,5,nSpcs);

LF(1);
TB(10); write('kFilm[i] (cm/s) = '); WV(kFilm,10,5,nSpcs);

LF(5)
```

```
end;  { SetParameters }


procedure RunSweep( tBegin, tEnd : RealNumber;
                    StepBegin : integer;
                    var StepEnd : integer );

var     Step : integer;
        CPUTime : integer;
        k : integer;

function VSurf( t : RealNumber ) : RealNumber;

begin

        VSurf := V[StepBegin] + b*(t - tBegin)

end;  { Vsurf }


function SurfBC( k : integer;
                 C, dCdx : VeCtor;
                 t : RealNumber ) : RealNumber;

var     V, Vohm : RealNumber;
        j : integer;


        function I : RealNumber;

        var     Sum : RealNumber;
                k : integer;

        begin
                Sum := 0;

                for k := 1 to nSpcs do
                        Sum := Sum - F*A*z[k]*D[k]*dCdx[k];

                I := Sum

        end;  { I }



        function iSurf( j : integer; V : RealNumber ) : RealNumber;
```

```
        var     k : integer;
                PiA, PiC, rA, rC : RealNumber;

        begin
                PiA := 1;
                PiC := 1;

                for k := 1 to nSpcs do
                begin

                        if ( p[k,j] > 0 ) then
                        begin
                            if ( C[k] = 0 ) then PiA := 0
                            else PiA := PiA*( (C[k]/Cref[k,j])**p[k,j] )
                        end;

                        if ( q[k,j] > 0 ) then
                        begin
                            if ( C[k] = 0 ) then PiC := 0
                            else PiC := PiC*( (C[k]/Cref[k,j])**q[k,j] )
                        end
                end;

                rA := PiA*exp( (alphaA[j]*F/(R*Temp))*(V - Uref[j]) );
                rC := PiC*exp( -(alphaC[j]*F/(R*Temp))*(V - Uref[j]) );

                iSurf := ioRef[j]*( rA - rC )

        end;  { iSurf }



        function FickFlux( k : integer ) : RealNumber;

        begin   FickFlux := -D[k]*dCdx[k]          end;


        function FaradayFlux( k : integer; V : RealNumber ) : RealNumber;

        var     Sum : RealNumber;
                j : integer;

        begin
                Sum := 0;

                for j := 1 to nRxns do
                        Sum := Sum - ( s[k,j]/(n[j]*F) )*iSurf(j,V);

                FaradayFlux := Sum

        end;  { FaradayFlux }

begin  { SurfBC }

        Vohm := I/(4*kappa*radius);

        V := VSurf(t) - Vohm;


        SurfBC := FickFlux(k) - FaradayFlux(k,V)  { = 0 }
```

```
end;  { SurfBC }



begin

        SuperPose( nSpcs, StepBegin, StepEnd, D, kFilm,
                          tBegin, tEnd, dt, C, dCdx, t,
                          Iterations, Err1, Err2, CPUTime,
                          SurfBC, MaximumIterations, Tolerance      );


        writeln; writeln;

        for Step := StepBegin to StepEnd do
        begin

                I[Step] := 0;

                for k := 1 to nSpcs do
                        I[Step] := I[Step] - F*A*( z[k]*D[k]*dCdx[k,Step] );

                V[Step] := VSurf(t[Step])

        end;


        PrintSweep(StepBegin, StepEnd, V, I,
                          Iterations, Err1, Err2, CPUTime)

end;  { RunSweep }



begin  { CycVolt }

        writeln; writeln;


        PrintTitle;
        ReadAndPrintParameters;
        SetParameters;


        for k := 1 to nSpcs do
        begin
                C[k,0] := Co[k];
                dCdx[k,0] := 0
        end;



        t[0] := tStart;
        V[0] := Vstart;
        I[0] := 0;

        Err1[0] := false;
```

```
        Err2[0] := false;
        Iterations[0] := 0;

        StartStep := 0;



        if ( b <> 0 ) then

            for Sweep := 1 to TotalSweeps do
            begin

                    tReversal := tStart + tRange;

                    RunSweep(tStart, tReversal, StartStep, ReversalStep);

                    tStart := tReversal;
                    StartStep := ReversalStep;
                    b := -b

            end

        else
            begin
                    LF(2); TB(20);
                    write('WARNING: Sweep Rate (b) = 0');
                    LF(1); TB(20);
                    write('==> dt undefined; PROGRAM EXECUTION HALTED.');
                    LF(5)
            end;


        LF(1); TB(20); write('Program End');  LF(1)

end  { CycVolt }
```

Sample Data File for CycVolt

```
------- Data File for CycVolt Program -------

nSpcs = 3
nRxns = 2

MaxIts = 10
Tolerance = 1e-8

D = < 2e-6, 2e-6, 2e-6 > cm2/s
z = < -1, 0, -1 >

Ure = 0 V

T = 298.15 K
rhoZero = 1e-3 kg/cm3
nu = 1e-2 cm2/s


Utheta = < 0.6, 1.265 > V

alphaA = < 1.6, 1.0 >
alphaC = < 0.4, 1.0 >
n = < 2, 2 >

ioRef = < 6.5e-5, 1e+5 > A/cm2

s = < <3, 0>, <0, -3>, <-1, 2> >

cRef = < <2.5e-6, 2.5e-6>, <2.5e-6, 2.5e-6>, <2.5e-6, 2.5e-6> > mol/cm3

Co = < 2.5e-6, 8.8e-23, 1.1e-17 > mol/cm3

Vstart = 0.4 V
Vreversal = 1.8 V

dV = 0.01 V

b = 0.2 V/s

radius = 0.1348 cm

kappa = 0.2 mho/cm

RotSpeed = 0 rpm

Sweeps = 2

-------- End of Data File ----------
```

Sample Output from CycVolt

Cyclic Voltammogram Program

Written by: Michael Matlosz
Last Update: February 19, 1985

Program begun at 14:31:52.76
on 14-MAR-1985

Parameters:

----- Input Data -----

Number of species (nSpcs) = 3
Number of reactions (nRxns) = 2

MaximumIterations = 10
Tolerance = 1.0000E-08

D[k] (cm2/s) = 2.0000E-06 2.0000E-06 2.0000E-06

z[k] = -1.0 0.00 -1.0

Ure (V) = 0.00000

T (K) = 298.15
rhoZero (kg/cm3) = 1.0000E-03
nu (cm2/s) = 0.010000

Utheta[j] (V) = 0.60000 1.2650

alphaA[j] = 1.6 1.0

alphaC[j] = 0.40 1.0

ioRef[j] (A/cm2) = 6.5000E-05 1.0000E+05

s[k,j] =

3.0000	0.00000
0.00000	-3.0000
-1.0000	2.0000

Cref[k,j] =

2.5000E-06	2.5000E-06
2.5000E-06	2.5000E-06
2.5000E-06	2.5000E-06

Co[k] = 2.5000E-06 8.8000E-23 1.1000E-17

Vstart (V) = 0.40000
Vreversal (V) = 1.8000
dV (V) = 0.010000
b (V/s) = 0.20000
radius (cm) = 0.13480
A (cm2) = 0.057086
kappa (mho/cm) = 0.20000
RotationSpeed (rpm) = 0.00000

TotalSweeps (forward and reverse) = 2

----- Derived Quantities -----

dt (s) = 0.050000

Uref[j] (V) = 0.75392 1.1880

p[i,j] =

3.0000	0.00000
0.00000	0.00000
0.00000	2.0000

q[i,j] =

0.00000	0.00000
0.00000	3.0000
1.0000	0.00000

omega (rad/s) = 0.00000
del[i] (cm) = undefined
kFilm[i] (cm/s) = 0.00000 0.00000 0.00000

Final Results: Anodic-direction Sweep

Step	V (Volts)	I (Amps)	Iterations
0	0.400000	0.000000	0
1	0.410000	-3.21392E-16	2
2	0.420000	2.56785E-16	2
3	0.430000	9.85697E-16	2
4	0.440000	2.39135E-15	2
5	0.450000	5.27127E-15	2
6	0.460000	1.12265E-14	2
7	0.470000	2.35267E-14	3
8	0.480000	4.88277E-14	3
9	0.490000	1.00601E-13	3
10	0.500000	2.05952E-13	3
11	0.510000	4.19068E-13	3
12	0.520000	8.47633E-13	3
13	0.530000	1.70435E-12	3
14	0.540000	3.40698E-12	3
15	0.550000	6.77150E-12	3
16	0.560000	1.33837E-11	3
17	0.570000	2.63105E-11	3
18	0.580000	5.14583E-11	3
19	0.590000	1.00156E-10	3
20	0.600000	1.94055E-10	3
21	0.610000	3.74403E-10	3
22	0.620000	7.19556E-10	4
23	0.630000	1.37797E-09	4
24	0.640000	2.63026E-09	4
25	0.650000	5.00569E-09	4
26	0.660000	9.50019E-09	5
27	0.670000	1.79833E-08	4
28	0.680000	3.39529E-08	6
29	0.690000	6.39228E-08	6
30	0.700000	1.19928E-07	6
31	0.710000	2.23904E-07	6
32	0.720000	4.14848E-07	6
33	0.730000	7.58992E-07	6
34	0.740000	1.35962E-06	6
35	0.750000	2.35334E-06	6
36	0.760000	3.86617E-06	6
37	0.770000	5.91356E-06	6
38	0.780000	8.30296E-06	6
39	0.790000	1.06657E-05	6
40	0.800000	1.26356E-05	6
41	0.810000	1.40122E-05	6
42	0.820000	1.47836E-05	6
43	0.830000	1.50524E-05	5
44	0.840000	1.49553E-05	5
45	0.850000	1.46174E-05	5
46	0.860000	1.41359E-05	5

47	0.870000	1.35802E-05	5
48	0.880000	1.29966E-05	5
49	0.890000	1.24150E-05	5
50	0.900000	1.18534E-05	5
51	0.910000	1.13222E-05	5
52	0.920000	1.08265E-05	5
53	0.930000	1.03681E-05	5
54	0.940000	9.94694E-06	5
55	0.950000	9.56178E-06	5
56	0.960000	9.21085E-06	5
57	0.970000	8.89221E-06	5
58	0.980000	8.60399E-06	5
59	0.990000	8.34463E-06	5
60	1.000000	8.11294E-06	5
61	1.01000	7.90826E-06	5
62	1.02000	7.73056E-06	5
63	1.03000	7.58054E-06	5
64	1.04000	7.45974E-06	5
65	1.05000	7.37070E-06	5
66	1.06000	7.31709E-06	5
67	1.07000	7.30382E-06	5
68	1.08000	7.33716E-06	5
69	1.09000	7.42473E-06	5
70	1.10000	7.57529E-06	5
71	1.11000	7.79824E-06	5
72	1.12000	8.10254E-06	5
73	1.13000	8.49501E-06	5
74	1.14000	8.97758E-06	5
75	1.15000	9.54375E-06	5
76	1.16000	1.01745E-05	5
77	1.17000	1.08352E-05	5
78	1.18000	1.14751E-05	5
79	1.19000	1.20326E-05	5
80	1.20000	1.24467E-05	5
81	1.21000	1.26734E-05	5
82	1.22000	1.26982E-05	5
83	1.23000	1.25405E-05	5
84	1.24000	1.22440E-05	5
85	1.25000	1.18614E-05	5
86	1.26000	1.14407E-05	5
87	1.27000	1.10175E-05	5
88	1.28000	1.06137E-05	5
89	1.29000	1.02406E-05	5
90	1.30000	9.90199E-06	5
91	1.31000	9.59737E-06	5
92	1.32000	9.32409E-06	5
93	1.33000	9.07863E-06	5
94	1.34000	8.85738E-06	5
95	1.35000	8.65694E-06	5
96	1.36000	8.47435E-06	5
97	1.37000	8.30704E-06	5
98	1.38000	8.15287E-06	5

99	1.39000	8.01003E-06	5
100	1.40000	7.87703E-06	5
101	1.41000	7.75263E-06	5
102	1.42000	7.63579E-06	5
103	1.43000	7.52565E-06	5
104	1.44000	7.42147E-06	5
105	1.45000	7.32264E-06	5
106	1.46000	7.22864E-06	5
107	1.47000	7.13900E-06	5
108	1.48000	7.05336E-06	5
109	1.49000	6.97136E-06	5
110	1.50000	6.89272E-06	5
111	1.51000	6.81718E-06	5
112	1.52000	6.74450E-06	5
113	1.53000	6.67448E-06	5
114	1.54000	6.60695E-06	5
115	1.55000	6.54173E-06	5
116	1.56000	6.47869E-06	5
117	1.57000	6.41769E-06	5
118	1.58000	6.35860E-06	5
119	1.59000	6.30133E-06	5
120	1.60000	6.24576E-06	5
121	1.61000	6.19180E-06	5
122	1.62000	6.13938E-06	5
123	1.63000	6.08841E-06	5
124	1.64000	6.03882E-06	5
125	1.65000	5.99054E-06	5
126	1.66000	5.94351E-06	5
127	1.67000	5.89768E-06	5
128	1.68000	5.85299E-06	5
129	1.69000	5.80938E-06	5
130	1.70000	5.76682E-06	5
131	1.71000	5.72525E-06	5
132	1.72000	5.68464E-06	5
133	1.73000	5.64495E-06	5
134	1.74000	5.60614E-06	5
135	1.75000	5.56817E-06	5
136	1.76000	5.53102E-06	5
137	1.77000	5.49466E-06	5
138	1.78000	5.45905E-06	5
139	1.79000	5.42416E-06	5
140	1.80000	5.38998E-06	5

Sweep End (Execution Time = 44880 milli-seconds)

Final Results: Cathodic-direction Sweep

Step	V (Volts)	I (Amps)	Iterations
140	1.80000	5.38998E-06	0
141	1.79000	5.35648E-06	5
142	1.78000	5.32363E-06	5
143	1.77000	5.29141E-06	5
144	1.76000	5.25980E-06	5
145	1.75000	5.22878E-06	5
146	1.74000	5.19833E-06	5
147	1.73000	5.16843E-06	5
148	1.72000	5.13907E-06	5
149	1.71000	5.11023E-06	5
150	1.70000	5.08190E-06	5
151	1.69000	5.05405E-06	5
152	1.68000	5.02667E-06	5
153	1.67000	4.99976E-06	5
154	1.66000	4.97329E-06	5
155	1.65000	4.94725E-06	5
156	1.64000	4.92164E-06	5
157	1.63000	4.89643E-06	5
158	1.62000	4.87163E-06	5
159	1.61000	4.84721E-06	5
160	1.60000	4.82316E-06	5
161	1.59000	4.79949E-06	5
162	1.58000	4.77617E-06	5
163	1.57000	4.75320E-06	5
164	1.56000	4.73057E-06	5
165	1.55000	4.70827E-06	5
166	1.54000	4.68629E-06	5
167	1.53000	4.66462E-06	5
168	1.52000	4.64325E-06	5
169	1.51000	4.62218E-06	5
170	1.50000	4.60139E-06	5
171	1.49000	4.58086E-06	5
172	1.48000	4.56060E-06	5
173	1.47000	4.54057E-06	5
174	1.46000	4.52076E-06	5
175	1.45000	4.50114E-06	5
176	1.44000	4.48167E-06	5
177	1.43000	4.46229E-06	5
178	1.42000	4.44292E-06	5
179	1.41000	4.42345E-06	5
180	1.40000	4.40371E-06	5
181	1.39000	4.38345E-06	5
182	1.38000	4.36234E-06	5
183	1.37000	4.33984E-06	5
184	1.36000	4.31519E-06	5
185	1.35000	4.28728E-06	5
186	1.34000	4.25448E-06	5

187	1.33000	4.21442E-06	5
188	1.32000	4.16368E-06	5
189	1.31000	4.09733E-06	5
190	1.30000	4.00843E-06	5
191	1.29000	3.88733E-06	5
192	1.28000	3.72108E-06	5
193	1.27000	3.49306E-06	5
194	1.26000	3.18352E-06	5
195	1.25000	2.77175E-06	5
196	1.24000	2.24096E-06	5
197	1.23000	1.58596E-06	5
198	1.22000	8.22059E-07	5
199	1.21000	-9.55617E-09	5
200	1.20000	-8.45073E-07	5
201	1.19000	-1.61169E-06	5
202	1.18000	-2.24635E-06	5
203	1.17000	-2.71041E-06	5
204	1.16000	-2.99432E-06	5
205	1.15000	-3.11270E-06	5
206	1.14000	-3.09475E-06	5
207	1.13000	-2.97487E-06	5
208	1.12000	-2.78587E-06	5
209	1.11000	-2.55547E-06	5
210	1.10000	-2.30514E-06	5
211	1.09000	-2.05039E-06	5
212	1.08000	-1.80172E-06	5
213	1.07000	-1.56580E-06	5
214	1.06000	-1.34651E-06	5
215	1.05000	-1.14580E-06	5
216	1.04000	-9.64391E-07	5
217	1.03000	-8.02255E-07	5
218	1.02000	-6.58985E-07	5
219	1.01000	-5.34030E-07	5
220	1.000000	-4.26874E-07	5
221	0.990000	-3.37139E-07	5
222	0.980000	-2.64669E-07	5
223	0.970000	-2.09564E-07	5
224	0.960000	-1.72203E-07	5
225	0.950000	-1.53233E-07	5
226	0.940000	-1.53527E-07	5
227	0.930000	-1.74102E-07	5
228	0.920000	-2.15977E-07	5
229	0.910000	-2.79960E-07	5
230	0.900000	-3.66348E-07	5
231	0.890000	-4.74549E-07	5
232	0.880000	-6.02681E-07	5
233	0.870000	-7.47233E-07	5
234	0.860000	-9.02982E-07	5
235	0.850000	-1.06336E-06	5
236	0.840000	-1.22136E-06	5
237	0.830000	-1.37085E-06	5
238	0.820000	-1.50783E-06	5

239	0.810000	-1.63120E-06	5
240	0.800000	-1.74270E-06	5
241	0.790000	-1.84614E-06	5
242	0.780000	-1.94650E-06	5
243	0.770000	-2.04904E-06	5
244	0.760000	-2.15876E-06	5
245	0.750000	-2.28020E-06	5
246	0.740000	-2.41736E-06	5
247	0.730000	-2.57378E-06	5
248	0.720000	-2.75261E-06	5
249	0.710000	-2.95666E-06	5
250	0.700000	-3.18839E-06	5
251	0.690000	-3.44983E-06	5
252	0.680000	-3.74249E-06	5
253	0.670000	-4.06716E-06	5
254	0.660000	-4.42365E-06	5
255	0.650000	-4.81054E-06	5
256	0.640000	-5.22490E-06	5
257	0.630000	-5.66197E-06	5
258	0.620000	-6.11497E-06	5
259	0.610000	-6.57495E-06	5
260	0.600000	-7.03090E-06	5
261	0.590000	-7.47004E-06	5
262	0.580000	-7.87842E-06	5
263	0.570000	-8.24185E-06	5
264	0.560000	-8.54700E-06	5
265	0.550000	-8.78269E-06	5
266	0.540000	-8.94108E-06	5
267	0.530000	-9.01858E-06	5
268	0.520000	-9.01632E-06	5
269	0.510000	-8.93990E-06	5
270	0.500000	-8.79870E-06	5
271	0.490000	-8.60464E-06	5
272	0.480000	-8.37076E-06	5
273	0.470000	-8.10990E-06	5
274	0.460000	-7.83364E-06	5
275	0.450000	-7.55165E-06	5
276	0.440000	-7.27146E-06	5
277	0.430000	-6.99848E-06	5
278	0.420000	-6.73634E-06	5
279	0.410000	-6.48718E-06	5
280	0.400000	-6.25203E-06	5

Sweep End (Execution Time = 49320 milli-seconds)

Program End

TECHNICAL INFORMATION DEPARTMENT
LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
BERKELEY, CALIFORNIA 94720