Title
Optimal Emergency Maneuvers Of Automated Vehicles

Authors
Shiller, Zvi
Sundar, Satish

Publication Date
1996

CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA. BERKELEY

Optimal Emergency Maneuvers of Automated Vehicles

Zvi Shiller
University of California, Los Angeles

California PATH Research Report
UCB-ITS-PRR-96-32

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

November 1996

ISSN 1055-1425

Optimal Emergency Maneuvers of Automated Vehicles

M.O.U. 125

Zvi S hiller

Department of Mechanical, Aerospace and Nuclear Engineering

University of California Los Angeles

Los Angeles, CA 90095

June 21, 1996

Abstract

This work addresses two issues related to emergency maneuvers of autonomous vehicles. The first concerns the time-optimal speeds along specified path, which can be used to design emergency maneuver. The second concerns the computation of optimal (shortest) lane-change maneuvers for on-line collision avoidance. To compute the time optimal trajectories, we first solve the inverse dynamics problem. The performance limits along a specified maneuver are then computed by minimizing the motion time along the path, assuming free initial and final speeds. This establishes the velocity limit curve, which serves as an upper bound for any feasible trajectory, including emergency maneuvers.

The optimal lane change maneuvers are computed by minimizing the longitudinal distance of a lane transition, assuming given initial and free final speeds, to yield the sharpest (and shortest) dynamically feasible maneuvers. The optimal maneuvers are used to determine the minimum distance beyond which an obstacle cannot be avoided, which when plotted in the phase

plane provides a valuable design tool for planning safe avoidance maneuvers. On-line generation of emergency maneuvers is also addressed using a simple point mass model and extrapolating from a nominal optimal maneuver. The maneuvers computed for the point mass model and the extrapolated maneuvers are shown to closely correlate with the optimal maneuvers computed for the planar bicycle model.

Keywords: vehicle dynamics, optimal control, emergency maneuvers, obstacle avoidance, safety.

Optimal Emergency Maneuvers of Automated Vehicles

MOU 125
Executive Summary

This report summarizes the two years project "Optimization tools for Intelligent Vehicles", MOU 125. The focus of this work was on developing strategies for generating emergency maneuvers. The vehicle is modeled as a planar rear-drive, front steering, vehicle, with three degrees-of-freedom in the plane, and two control inputs: the steering angle and the rear tractive force. The tire side forces are approximated by a piecewise linear saturation function, and are coupled to the longitudinal force by the friction ellipse, derived from the Dugoff model.

The emergency maneuvers were computed in two steps. First, the time-optimal velocity profile along a specified path was maximized to yield the vehicle's performance envelope. Minimizing motion time is a suitable frame work for computing emergency maneuvers because, in addition to minimizing the disturbance to the traffic flow, time optimal trajectories represent the ultimate vehicle's performance between given boundary conditions, which might be needed in true emergency conditions.

To compute the time optimal trajectories, we first solve the inverse dynamics problem, i.e. compute the nominal (open loop) control inputs that would drive the vehicle along a specified path at given speeds. Parameterizing the path by its arc length, we derive the constraints on the vehicle's

speeds and accelerations due to the saturation of the tire forces and path geometry. The performance limit along a specified maneuver were then determined in the form of limiting trajectories by solving an optimal control problem that minimizes the motion time with free initial and final speeds. The optimization problem is solved as a parameter optimization, using exact penalty functions to account for the control constraints. The optimal trajectories, which are shown to slide along the control and the geometric path constraints, can be used for planning emergency maneuvers, and for verifying that a given trajectory is dynamically feasible, and hence dynamically safe.

The second step was to compute optimal lane-change maneuvers for collision avoidance. These maneuvers were computed by minimizing the longitudinal distance of a lane transition, assuming given initial and free final speeds, to yield the sharpest (and shortest) dynamically feasible maneuvers. The optimal maneuvers can be used as nominal trajectories for on-line feedback controllers. They can also be used to determine the minimum distance beyond which an obstacle cannot be avoided, which when plotted in the phase plane provides a valuable design tool for planning safe avoidance maneuvers. This plot clearly represents the distance to the obstacle, as a function of vehicle's speed, beyond which a lane change maneuver should not be attempted, but a head-on collision should be chosen instead.

On-line generation of emergency maneuvers is also addressed using a simple point mass model and extrapolating from a nominal optimal maneuver. The maneuvers for the point mass model and the extrapolated maneuvers are shown to closely correlate with the optimal maneuvers computed for the planar bicycle model. They can be computed on-line, taking into account

the vehicle's actual speed and road conditions.

A 6D dynamic model was also developed for dynamic simulations of the computed trajectories. It consists of the rigid body dynamics, suspensions, and the tire forces, while ignoring drive train dynamics. This model was used to test, the optimal trajectories computed in the course of this project, with inconclusive results. Further study into the dynamic simulation was out of scope of this two year project.

Contents

Chapter 1

Introduction

This project has addressed two main issues related to safety and emergency maneuvers of automated vehicles. The first is the issue of the vehicle's performance envelope for motions along specified paths. Establishing the performance envelope is essential for ensuring that the vehicle traverses safely challenging maneuvers, such as tight turns and lane transitions. Automatically satisfying the vehicle's performance envelope will significantly reduce accidents of passenger cars and big tracks, usually attributed to poor judgment of vehicle speeds and driver fatigue while approaching a sharp turn or changing lanes.

The second issue is obstacle avoidance at high speeds. Collision avoidance represents a central safety issue for automated vehicles. It concerns the avoidance of static obstacles, such as disabled vehicles and large objects blocking the forward path. Collision can be avoided either by decelerating to a full stop without hitting the obstacle, or by executing a timely lane-change

maneuver. A lane-change maneuver is generally more desirable since it may least disturb the traffic flow. However, the exact maneuver might depend on the distance to the obstacle, vehicle speed, and the traffic in the neighboring lanes. Of particular interest are emergency lane-change maneuvers, i.e. those that reach the vehicle's performance limit.

Vehicle control has been traditionally studied for separate longitudinal or lateral motions. Extensive studies have been conducted on the longitudinal control problem, which concerns the control of the longitudinal speed or displacement using the engine torque as the control input (Peng and Tomizuka 1990. Sheikholeslam and Desoer 1992, Godbole and Lygeros 1994). Other studies have focused on lateral control, which concerns the control of the yaw motion, assuming a constant longitudinal speed, using the steering angle and the engine torque as the control inputs (Peng and Tomizuka 1991, Hessburg and Tomizuka 1991, Hessburg et. al. 1991, Chee and Tomizuka 1994). Fewer studies have addressed the combined longitudinal and lateral motion problem (Narendran and Hedrick 1993).

In (Peng and Tomizuka 1991, Hessburg and Tomizuka 1991, Hessburg et. al. 1991), optimal preview control and a fuzzy logic controller were developed for lane following, based on a linearized vehicle model, and considering lateral motions only. In (Chee and Tomizuka 1994), these controllers were modified for lane change maneuvers, and were tested in simulations for a lane change at 110 km/h. While this speed might be considered extreme, the lane transition was rather long, lasting 6s over $180m$[1] . In (Narendran and Hedrick 1993), a

[1] In our present work, the vehicle traverses a lane change transition of **60m** in less than **2s**

sliding mode controller was developed for lane change maneuvers, considering a combined lateral and longitudinal vehicle model, and tested in numerical simulations at nominal conditions.

Though satisfactory for moderate conditions, controllers based on linear models were found ineffective for emergency maneuvers, at which simultaneous steering and braking are required at high lateral accelerations (Smith and Starkey 1994). In that study, a controller for emergency maneuvers was developed by optimizing the gains of a linear controller over the step response of the nonlinear model. The resulting controller was tested successfully in simulations at $30m/s$ (about 70 MPH) over a $60m$ lane transition. However: the controller gains are path and speed specific, which hinders the applicability of this method for general emergency maneuvers. Clearly, the emergency maneuvering problem is far from solved, and a better understanding of the vehicle's performance at high speeds is needed before effective controllers for emergency conditions can be developed.

In this work, we first address the problem of emergency maneuvers in the context. of time optimal control. Specifically, we compute time optimal trajectories along specified paths for a nonlinear vehicle model, which considers both lateral and longitudinal motions. Minimizing motion time minimizes the disturbance to the traffic flow, in addition to establishing the ultimate vehicle's performance between given boundary conditions.

To compute time optimal trajectories, it is necessary to first solve the inverse dynamics problem, i.e., compute the nominal (open loop) control inputs required to traverse a specified trajectory. For this task, we consider ground vehicles with front steering and a rear-drive (and brake), modeled

in the plane with three degrees-of-freedom, using the steering angle and the rear tractive force as the two control variables. The vehicle is assumed to move at given speeds along a specified path. Since this specifies only the positions of the mass center, or two of the vehicle's three degrees-of-freedom in the plane, the vehicle's orientation remains to be determined.

Writing the equations of motion in the body frame, it is shown that the vehicle's orientation is constrained by the moment equation, which is a second order differential equation in the body slip angle (the angle between the body axis and the velocity of its mass center). Numerically integrating this equation yields the vehicle's orientation, which depends on path geometry and on vehicle parameters and speeds. The steering angle and the tractive force along the specified path are computed from the two force equations, once the body slip angle along the path has been computed. The kinematic solution for the body slip angle, assuming zero speeds and no tire slip, is also developed, requiring the numerical integration of a first order differential equation. It can be used to predict vehicle behavior at low speeds.

The vehicle's performance limits are determined in the form of limiting trajectories by solving an optimal control problem that minimizes the motion time with free initial and final speeds. The optimization problem is solved as a parameter optimization, using exact penalty functions to account for the control constraints. This essentially maps the control constraints to the trajectory space. The optimal trajectories, which are shown to slide along the control and the geometric path constraints, represent the upper bounds on vehicle's speeds. They can be used for planning emergency maneuvers, and for verifying that a given trajectory is dynamically feasible, and hence

dynamically safe. This approach to computing the performance limits is more constructive than previous work, which establishes the performance limits by simulating vehicle dynamics along various maneuvers, then observing the control inputs that result in spin or skid (Allen et. al. 1991).

We also compute optimal lane-change maneuvers that minimize the lateral displacement of the lane transition, assuming given initial and free final speeds. These are true emergency maneuvers since they minimize the reaction time and the reaction distance.

The optimal maneuvers represent the sharpest feasible maneuver at a given speed. They define the minimum cleating distance, the distance beyond which an obstacle cannot be avoided by a lane-change maneuver at given initial speeds. Plotting the minimum clearing distance in the phase-plane produces a performance limit curve, which provides a valuable decision tool for planning safe avoidance maneuvers.

The optimal maneuvers can be used as the reference input to on-line feedback controllers. Since the computation of the optimal maneuvers is too time consuming for on-line applications, we propose two on-line alternatives. The first approximates the optimal maneuvers using a simple point mass model. The second extrapolates an emergency maneuver for a given initial speed from a stored optimal maneuver computed for some nominal speed. Both are shown to be computationally very efficient and to produce trajectories that closely correlate with the optimal maneuvers computed for the planar vehicle model.

A 6D dynamic model was developed for dynamic simulations of the computed trajectories. It consists of the rigid body dynamics. suspensions, and

the tire forces, while ignoring drive train dynamics*. This model was used to test the optimal trajectories computed in the course of this project, with inconclusive results. Further study into the dynamic simulation was out of scope of this two year project.

This work was summarized in (Shiller and Sundar 1995, Shiller and Sundar 1996). The optimization routines, enhanced with computer graphics, were implemented in FORTRAN and C on a Silicon Graphics workstation.

This report is organized as follows. Chapter 1 presents the computation of the velocity limit curve along specified paths, to be used as an upper limit for emergency maneuver. Chapter 2 presents the off-line and on-line computation of the optimal lane change maneuvers, to be used for obstacle avoidance. Chapter 3 presents the derivation of the 6D dynamic model.

[2] We found a need to develop this model since we could not use the dynamic model used by PATH: it includes the drive train and tire dynamics, which prevented us from specifying the rear tractive force as a control input. The source code was also difficult to modify to our needs.

Chapter 2

Performance Envelope

In this chapter, we present the computation of the time-optimal velocity profiles along specified paths (Shiller and Sundar 1995).

2.1 Vehicle Model: The Bicycle Model

In selecting the vehicle model, we evaluated the complex model developed by Lunger and adopted by Tomizuka (Peng and Tomizuka 1990). Because of the large number of states used in this model, we decided to consider first the planar bicycle model. It ignores the load shift due to roll and pitch, but it accounts for the side forces of the front and rear wheels, which are nonlinear functions of the front and rear slip angles, respectively. To practically solve the inverse dynamics problem, we approximated the tire forces by a piece-wise linear saturation function of the slip angle. The two control inputs are assumed to be the front steering angle and the rear tractive/braking force.

The front wheel is assumed to provide only a side force, and no tractive force or resistance. This model has only six independent states: the position of the mass center in the plane, vehicle's orientation, and their time derivatives.

We first derive the kinematic relations between the slip angles of the front and rear wheels and the steering angle of the planar bicycle model shown in figure 2.1, with front steering and rear driving (and braking). The absolute orientation of the vehicle in the inertial frame is φ; the angle of the linear velocity of its mass center is θ; the angle of the linear velocity of its mass center, in the body frame, is α; and the steering angle of the front wheel in the body frame is β. All angles are positive in the counter clockwise direction.

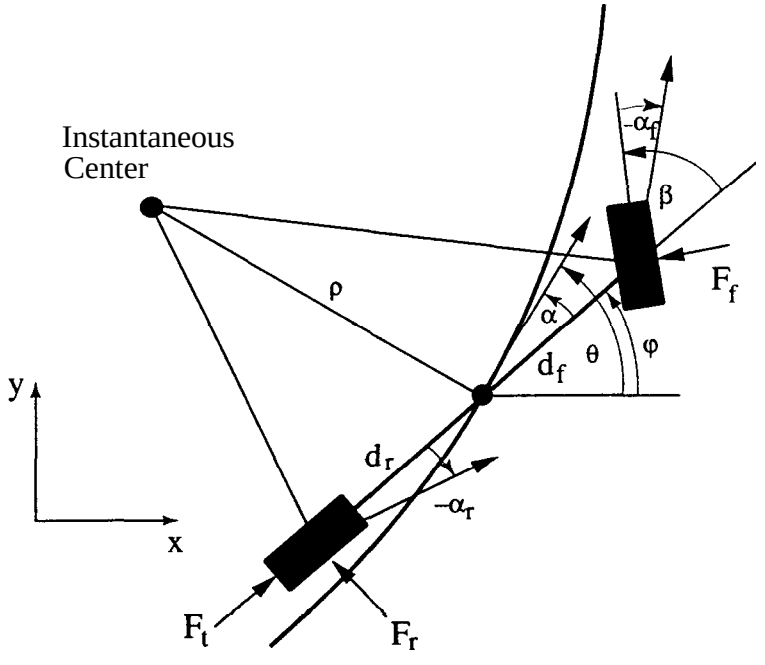

Figure **2.1:** The bicycle model

Figure 2.1 shows the vehicle at a typical point along a specified path. The velocity of the mass center coincides with the path tangent and is at an angle θ relative to the inertial frame. Thus,

$$\varphi = \theta - \alpha \tag{2.1}$$

For general motions, the vehicle undergoes pure rotation around some instantaneous center, located at ρ (the instantaneous radius) from the mass center, as shown in figure 2.1. Note that $\rho > 0$ for a counterclockwise turn, and $\rho < 0$ for a clockwise turn.

The linear velocity of every point on the vehicle is perpendicular to the line from that point to the instantaneous center. The angles between the linear velocities of the rear and front wheels and their major axes are the rear and front slip angles, α_r and α_f, shown in figure 2.1. Consistent with this notation, we call α the body *slip angle.*

The wheel slip angles can be derived as explicit functions of α and the instantaneous curvature:

$$\alpha_r = tan^{-1}\left[\frac{d_r\kappa + \sin\alpha}{\cos\alpha}\right] \tag{2.2}$$

$$\alpha_f = \gamma - \beta \tag{2.3}$$

where

$$\gamma = tan^{-1}\left[\frac{d_f\kappa + \sin\alpha}{\cos\alpha}\right], \tag{2.4}$$

d_r and d_f are the lengths of the vehicle between the mass center and the centers of the rear and front wheels, respectively, and $\kappa = 1/\rho$ is the instantaneous curvature. Henceforth, we will denote ρ and κ as the instantaneous

radius and curvature, and R and K as the radius and curvature corresponding to the specified path. An equivalent representation of the slip angles, in terms of the Cartesian coordinates, is

$$\alpha_r = \tan^{-1}\left[\frac{-d_r\dot{\varphi} + \dot{y}\cos\varphi - \dot{x}\sin\varphi}{\dot{x}\cos\varphi + \dot{y}\sin\varphi}\right] \tag{2.5}$$

$$\alpha_f = tan^{-1}\left[\frac{d_f\dot{\varphi} + \dot{y}\cos\varphi - \dot{x}\sin\varphi}{\dot{x}\cos\varphi + \dot{y}\sin\varphi}\right] \tag{2.6}$$

where $\dot{x}$ and $\dot{y}$ represent the velocity of the mass center in the inertial frame.

For transient motions, the instantaneous center does not necessarily coincide with the center of path curvature. This can be seen by recognizing that the instantaneous center of the rotating body satisfies

$$\frac{1}{\rho} = \kappa = \frac{\dot{\varphi}}{\dot{s}} = \varphi_s = \theta_s - \alpha_s \tag{2.7}$$

where the subscript s denotes derivatives with respect to $s \in [0, s_f]$, the path arc length. Similarly, the center of path curvature satisfies

$$\frac{1}{R} = K = \theta_s \tag{2.8}$$

Hence, path curvature coincides with the curvature of the rotating body if and only if $a_s = 0$, i.e. α = constant. The two are identical for steady-state motions since then the slip angle, α, is constant.

The external forces acting on the vehicle consist of the tire forces shown in figure 2.1. The front tire generates only a side force, F_f, since the front wheel is assumed to rotate freely. The rear tire generates both side. F_r, and longitudinal, F_t, forces due to the engine torque or brakes. Treating F_t as a control input, we ignore motor and drive train dynamics. The actual engine

input can later be computed for the resulting tractive force, using the inverse dynamics of the tire and drive train subsystem.

We can write the equations of motion in the inertial frame, to yield:

$$F_t \cos \varphi - F_r \sin \varphi - F_f \sin(\varphi + \beta) = m\ddot{x} \tag{2.9}$$

$$F_t \sin \varphi + F_r \cos \varphi + F_f \cos(\varphi + \beta) = m\ddot{y} \tag{2.10}$$

$$-d_r F_r + d_f F_f \cos \beta = I+ \tag{2.11}$$

where m is the vehicle mass, and I is its moment of inertia around the mass center. These equations are coupled in the control inputs F_t and β. To simplify the computation of the control inputs, we rederive the equations of motion in the body frame, and express the moment equation around the center of the front wheel:

$$F_t - F_f \sin(\beta) = \text{ma}, \tag{2.12}$$

$$F_r + F_f \cos(\beta) = ma_y \tag{2.13}$$

$$-(d_f + d_r)F_r = I\ddot{\varphi} - md_f a_y \tag{2.14}$$

where a, and a_y are the projections of the absolute acceleration of the mass center on the body frame.

This formulation partially decouples the equations of motion in terms of the controls: the moment equation is not explicit in the controls; the force equation in the x direction includes only F_t, and the remaining force equation includes both F_t and β. This formulation also makes clear that the angular acceleration, $\ddot{\varphi}$, cannot be arbitrarily specified, since F_r is a state dependent force. The moment equation (2.14) is, therefore, an equality constraint that

eliminates one of the vehicle's three degrees-of-freedom. It will be used to compute the vehicle's orientation along a specified path.

2.2 Tire model

We have studied several tire models, including those developed by Dugoff (1970) and Sakai (1981). The Dugoff model is an empirical model, providing analytical relations for the longitudinal and lateral forces as functions of the slip angle and slip ratio. It thus accounts for the coupling between the side and longitudinal forces, known as the friction *ellipse.*

The Dugoff model for a typical tire is (Dugoff 1970, Wong, 1978)[1]

$$F_x = -\frac{C_s s}{1-s} f(\lambda) \tag{2.15}$$

$$F_y = -\frac{C_\alpha \tan\alpha}{1-s} f(\lambda) \tag{2.16}$$

where

$$\lambda = \frac{\mu W (1-\epsilon v\sqrt{s^2 + \tan^2\alpha})(1-s)}{2\sqrt{c_s^2 s^2 + C_\alpha^2 \tan^2\alpha}} \tag{2.17}$$

$$f(\lambda) = \begin{cases} X(2-\lambda) & \text{for } \lambda < 1 \\ 1 & \text{for } \lambda > 1 \end{cases} \tag{2.18}$$

and

F_x = longitudinal force

F_y = side force

[1] In this section, we use α to denote the tire slip angle

s = longitudinal slip

α = tire slip angle

C_s = cornering stiffness

C_α = longitudinal stiffens

W = vertical load

v = vehicle speed

μ = coefficient of road adhesion

ϵ = adhesion reduction coefficient

Figure 2.2 shows the longitudinal and side forces for various slip angles at fixed tire parameters. These curves can be approximated by ellipses, with major axes dependent on the slip angle:

$$(\frac{F_x}{F_{xmax}})^2 + (\frac{F_y}{F_{ymax}})^2 = 1 \tag{2.19}$$

F_{ymax} is computed by setting the slip ratio s = 0 in (2.16), and F_{xmax} is approximated by:

$$F_{xmax} = \lim_{s \to 1} F_x = \frac{C_s \mu W (1 - \epsilon v \sqrt{1 + \tan^2 \alpha^*})}{\sqrt{C_s^2 + C_\alpha^2 \tan^2 \alpha^*}} \tag{2.20}$$

The Sakai model is theoretical, based on modeling of the tire/road interaction (Sakai 1981). The longitudinal and side forces computed by this model are coupled by elliptical curves which better approximate the experimental data than the Dugoff model. However, the Dugoff model is simpler, and its friction ellipse easy to compute (Maalej et. al. 1989). In addition, the Dugoff model underestimates the actual tire forces, which might be beneficial for computing safe emergency maneuvers.

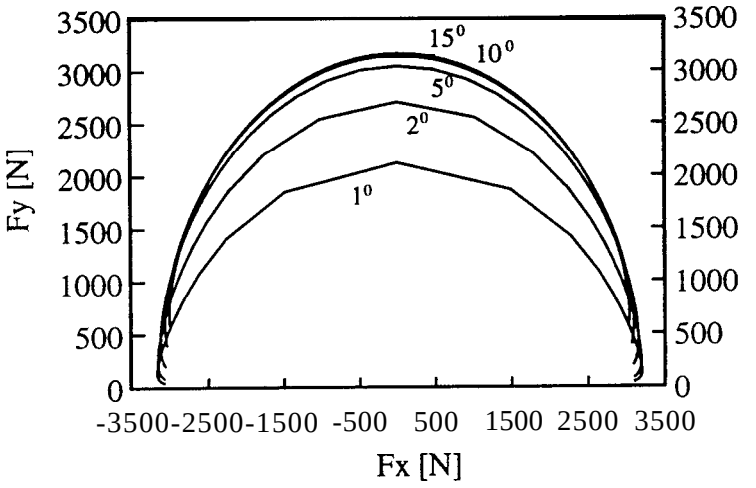

Figure 2.2: The Dugoff model.

Using either tire model would severely couple the equations of motion with the tractive force, making the inverse dynamics problem difficult to solve. For simplicity, we, therefore, ignore the friction ellipse at low slip angles and model the lateral force by the piecewise linear saturation function

$$F_r = \begin{cases} -C_\alpha \alpha & \text{if } |\alpha| < \alpha^* \\ -C_\alpha \alpha^* & \text{if } \alpha \geq \alpha^* \\ C_\alpha \alpha^* & \text{if } \alpha \leq -\alpha^* \end{cases} \qquad (2.21)$$

where α^* is the critical tire slip angle, corresponding to the maximum side force. The tractive force, F_t, is generated by the forward slip of the rear wheel, independent of the slip angle. To account for the coupling between the lateral and longitudinal forces, we bound the vector sum of the tractive

and the side forces by the friction ellipse that is fixed at the critical slip angle

$$(\frac{F_x}{F^*_{xmax}})^2 + (\frac{F_y}{F^*_{ymax}})^2 \leq 1 \tag{2.22}$$

where F^*_{xmax} and F^*_{ymax} are computed at the critical slip angle $\alpha = \alpha^*$.

Here, the vertical load, W, is constant due to the assumption of a planar bicycle model.

2.3 Inverse Dynamics

To solve the inverse dynamics problem: we substitute the tire model (2.21) (only the first linear segment) into the equations of motion, to yield

$$F_t + C_f \alpha_f \sin(\mathrm{P}) = m a_x \tag{2.23}$$

$$F_r - C_f \alpha_f \cos(\beta) = m a_y \tag{2.24}$$

$$(d_f + d_r) C_r \alpha_r = I\ddot{\varphi} - m d_f a_y \tag{2.25}$$

where C_r and C_f are the coefficients of the cornering stiffness of the rear and front tires, respectively. Since both, F_f and F_r, are functions of the body slip angle, α, it is necessary to first determine the vehicle's orientation for given path and speeds, using (2.25). We can then compute β, using (2.24). and F_t, using (2.23). The vehicle's path and speeds are defined by the time histories of the location, x(t), of the mass center.

The computation of α is first presented for kinematic conditions, assuming zero slip. For curved motions, this is equivalent to assuming zero speeds, since it is only at zero speeds that the vehicle can follow a curved path with zero lateral tire forces.

2.3.1 Kinematic α

The assumption of zero slip implies that the velocities of the rear and front wheels coincide with their major axes, or $\alpha_r = \alpha_f = 0$, as shown in figure 2.3. It follows that at every point along the path

$$\rho \sin(\alpha) = d_r \tag{2.26}$$

Using (2.7) we obtain a first order differential equation in α:

$$\alpha_s(s) + \frac{1}{d_r} \sin(\alpha(s)) = \theta_s(s) \tag{2.27}$$

Note that the forcing function of (2.27) is path curvature, $\theta_s(s)$.

We can solve for the kinematic α, α_k, by numerically integrating (2.27), starting at known initial conditions. We can then solve for the corresponding steering angle, β_k, using the trigonometric relation (see figure 2.3):

$$tan(\beta_k) = \frac{d_r + d_f}{d_r} tan(\alpha_k) \tag{2.28}$$

The kinematic solution for α and β might be used as close approximations of the transient solution at low speeds.

Note that any α other than α_k would result in a nonzero rear slip angle, α_r. Assuming a counter clockwise turn, we obtain that $\alpha < \alpha_k \rightarrow \alpha_r < 0$, and $\alpha > \alpha_k \rightarrow \alpha_r > 0$. Therefore, the vehicle will spin if $\alpha < \alpha_k$ along a counter clockwise turn, since then the direction of the rear force is opposite of the instantaneous center.

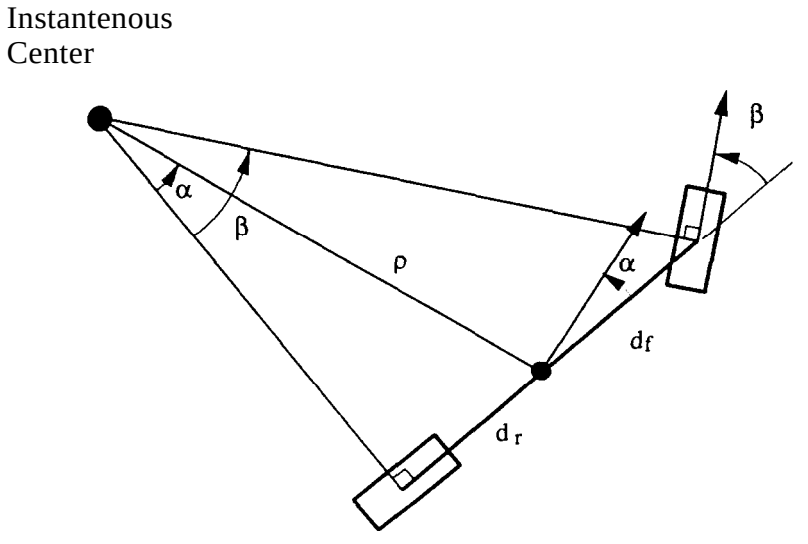

Figure 2.3: Kinematic motion

2.3.2 **Dynamic** α

To compute the dynamic (transient) α, we first represent the velocity and acceleration of the mass center in terms of the speed and acceleration along the path. This is done by parameterizing the path by the path arc length, $s \in [0, s_f]$, so that any point along the specified path is represented by the vector function x(s) = {z(s), y(s)}. Differentiating x once with respect to s yields the unit tangent $\mathbf{x}_s$; differentiating twice yields the normal $\mathbf{x}_{ss}$ of magnitude $||\mathbf{x}_{ss}|| = K$, where $||\cdot||$ denotes the Euclidean norm. The velocity, $\mathbf{v}_c$, and acceleration, a,, of the center of mass, can be expressed in terms of the speed, $\dot{s}$, and acceleration, $\ddot{s}$, tangent to the path:

$$\mathbf{v}_c = \dot{s}\mathbf{x}_s \tag{2.29}$$

$$\mathbf{a}_c = \ddot{s}\mathbf{x}_s + \dot{s}^2\mathbf{x}_{ss} \tag{2.30}$$

It is easy to show that the components of the absolute acceleration (2.30) in the body frame, are:

$$a, = \ddot{s}\cos(\alpha) - \dot{s}^2 K \sin(\alpha) \tag{2.31}$$

$$a_y = \ddot{s} \sin(o) + \dot{s}^2 K \cos(\alpha) \tag{2.32}$$

The angular velocity and acceleration can also be expressed in terms of the speed and tangent acceleration:

$$\dot{\theta} - \dot{\alpha} = (6, - \alpha_s)\dot{s} \tag{2.33}$$

$$\ddot{\theta} - a = (\theta_s - \alpha_s)\ddot{s} + (\theta_{ss} - \alpha_{ss})\dot{s}^2 \tag{2.34}$$

We can now write (2.25) as a second order differential equation in α:

$$(d_r + d_f)C_r\alpha_r = I[(\theta_s - \alpha_s)\ddot{s} + (\theta_{ss} - \alpha_{ss})\dot{s}^2] - md_f[\ddot{s}\sin(\alpha) + \dot{s}^2\theta_s\cos(\alpha)] \tag{2.35}$$

where

$$\alpha_r = tan^{-1}[\frac{-d_r(\theta_s - a,) + \sin(\alpha)}{cos(a)}] \tag{2.36}$$

Solving for α_{ss} yields

$$\alpha_{ss} = -(d_f + d_r)\alpha_r(\alpha)\frac{C_r}{I\dot{s}^2} - \frac{m}{I}d_f\frac{\ddot{s}}{\dot{s}^2}\sin(\alpha) - \alpha_s\frac{\ddot{s}}{\dot{s}^2} + (\frac{\ddot{s}}{\dot{s}^2} - \frac{m}{I}d_f\cos(\alpha))\theta_s + \theta_{ss} \tag{2.37}$$

We can compute Q(s) by numerically integrating α_{ss} from known initial conditions, using θ_s and θ_{ss} as the forcing functions. It is convenient to integrate from steady-state motion along straight line segment, for which a(s) = $\alpha_s(s)$ = 0.

Note that θ_{ss} is defined only for continuous path curvatures. At points where the curvature is discontinues, α_s is also discontinuous. To compute

the discontinuity in α_s, we integrate $\theta_{ss} - \alpha_{ss}$ with respect to s across the discontinuity

$$\int_{s_-}^{s+} (\theta_{ss} - \alpha_{ss})ds = (\theta_s(s) - \alpha_s(s))]_{s_-}^{s+} = \varphi_s]_{s_-}^{s+} = \frac{\dot{\varphi}}{\dot{s}}]_{s_-}^{s+} = 0 \qquad (2.38)$$

This integral equals zero since $\dot{\varphi}$ and $\dot{s}$ are continuous everywhere. Therefore,

$$\alpha_s(s_+) = \alpha_s(s_-) + \Delta\theta_s(s) \qquad (2.39)$$

The jump in α_s is, thus, equal to the jump in θ_s.

Figure 2.5 shows the dynamic α for motions at various constant speeds along the path shown in figure 2.4. This path consists of a B-spline, which is C^2 continuous. As a result, α is smooth and differentiable for all speeds. The kinematic α, shown in figure 2.5 as a dotted line, was computed using *(2.27)*.

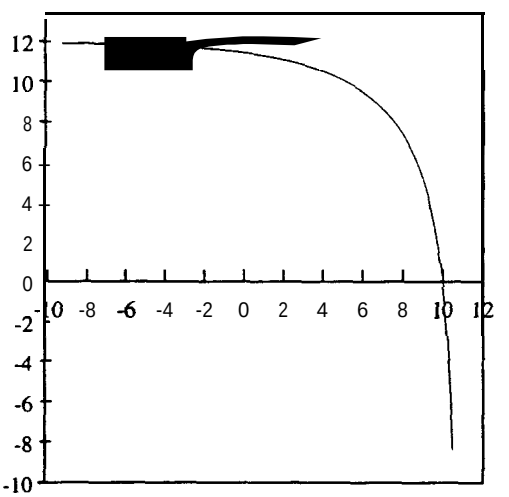

Figure 2.4: A B spline path

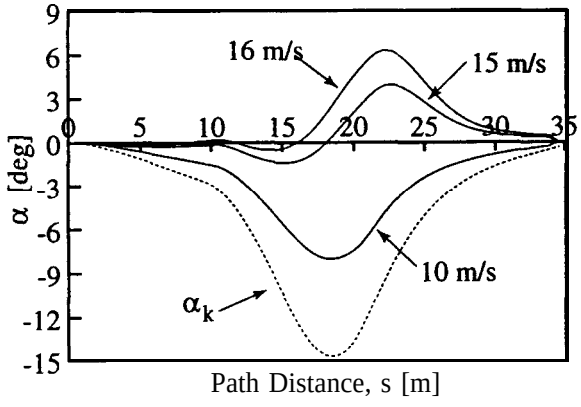

Figure 2.5: Dynamic α for various speeds along the B spline path

Figure *2.7* shows the dynamic α for the path shown in figure 2.6. consisting of two straight line segments, connected by a circular arc. Clearly, the curvature along this path is discontinuous across the junctions between the straight lines and the circular arc. At these points, α_s (the slope of α) is also discontinuous, as can be seen in figure 2.7. Here, α exhibits a typical response of an under-damped second order system as it oscillates, following each step in the curvature. These oscillations depend on the vehicle speed, as can be explained by substituting into (2.37) $\ddot{s} = 0$, assuming small α and α_r, and considering a straight line path (6, = $\theta_{ss} = 0$), to yield:

$$\alpha_{ss} + \frac{C_r d_r (d_f + d_r)}{I\dot{s}^2}\alpha_s + \frac{C_r (d_f + d_r)}{I\dot{s}^2}\alpha = 0 \qquad (2.40)$$

Clearly, (2.40) represents a second order system in α with a damping ratio, ζ, inversely proportional to $\dot{s}$:

$$\zeta = \sqrt{\frac{C_r d_r^2 (d_f + d_r)}{4I\dot{s}^2}} \qquad (2.41)$$

Therefore, α would oscillate as a result of a nonzero forcing function or nonzero initial conditions. The higher the speed, the larger the oscillations,

as we have observed. Also, the damping ratio is proportional to d_r, the distance of the mass center from the rear axle. Therefore, the closer the center of mass to the front axle, the more damped the oscillations.

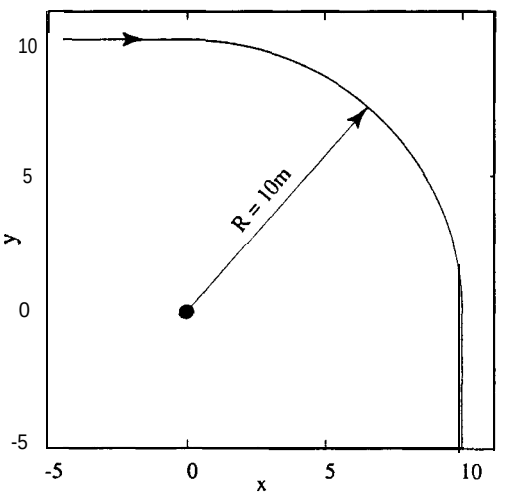

Figure 2.6: A Circular turn

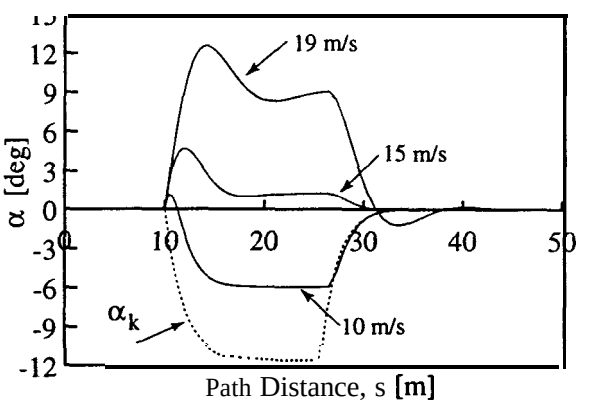

Figure 2.7: Dynamic a along the circular arc

2.3.3 Steady-State α

To gain insights into the behavior of the dynamic α, we investigate steady-state motions along a circular arc, along which α and θ_s are constant. With these assumptions, (2.35) reduces to

$$\alpha_r = \frac{-md_f\dot{s}^2\theta_s}{(d, + d_f)C_r}\cos(\alpha) \qquad (2.42)$$

where

$$\alpha_r = tan^{-1}[\frac{-d_r\theta_s + \sin(o)}{\cos(\alpha)}] \qquad (2.43)$$

To solve for the steady-state α, we intersect (2.42) and (2.43), as shown schematically in figure 2.8 for two values of $\dot{s}$ along a counter clockwise turn ($\theta_s = \kappa > 0$). For nonzero $\dot{s}$, (2.42) is a cosine with an amplitude proportional to $\dot{s}^2$, whereas (2.43) is close to a straight line, crossing the horizontal axis at α_k. For this turn, as $\dot{s}$ increases, the amplitude of the cosine curve increases, moving the intersection point left of α_k. Therefore, α decreases with speed along the counter clockwise turn, changing from a positive value (the rear wheel inside the curve) at low speeds to a negative value (the rear wheel outside the curve) at high speeds, as shown in figure 2.9. Also shown in figure 2.9 is the steady-state β, which is discussed next.

Driving at a negative α, referred to as "drifting a car through a bend", is common in competitive driving of rear-driven cars (Frere 1992). This type of driving is better observed on loose or slippery ground since the amplitude of the cosine in (2.42) is inversely proportional to the cornering stiffens, C,. A lower stiffens, or a larger amplitude, makes the variations in α with respect to $\dot{s}$ more pronounced.

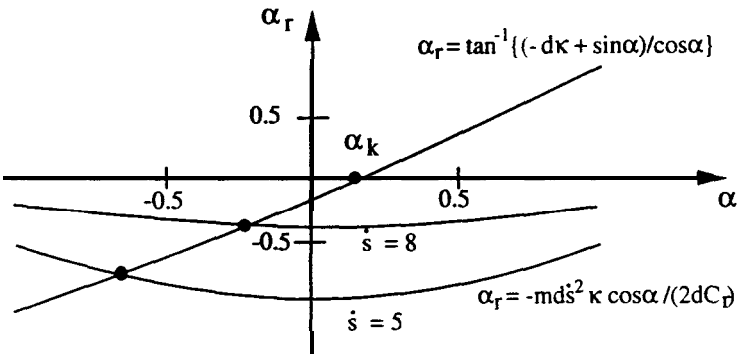

Figure 2.8: Solving for the steady-state α

2.3.4 Computing β

Once α is known, we can use (2.13) to solve for the steering angle: β. Rewriting (2.13) and using (2.32), yields

$$-C_r\alpha_r - C_f(tan^{-1}[\frac{d_f\kappa + \sin(\alpha)}{\cos(\alpha)}] - \beta)\cos(\beta) = m(\ddot{s}\ \sin(a)\ +\ \dot{s}^2 K\cos(\alpha)) \tag{2.44}$$

To solve for β, we rewrite (2.44) as

$$(a_1 - \beta)\cos(\beta) = a_2 \tag{2.45}$$

where

$$a_1 = tan^{-1}[\frac{d_f\kappa + \sin(\alpha)}{\cos(\alpha)}] \tag{2.46}$$

$$a_2 = -\frac{C_r}{C_f}\alpha_r - \frac{m}{C_f}(\ddot{s}\sin(\alpha) + \dot{s}^2 K\cos(\alpha)) \tag{2.47}$$

Plotting both sides of (2.45) reveals two possible solutions in the range $\beta = [-\pi/2, \pi/2]$, as shown in figure 2.10 for $a_1 = 0.1$. We choose the solution that

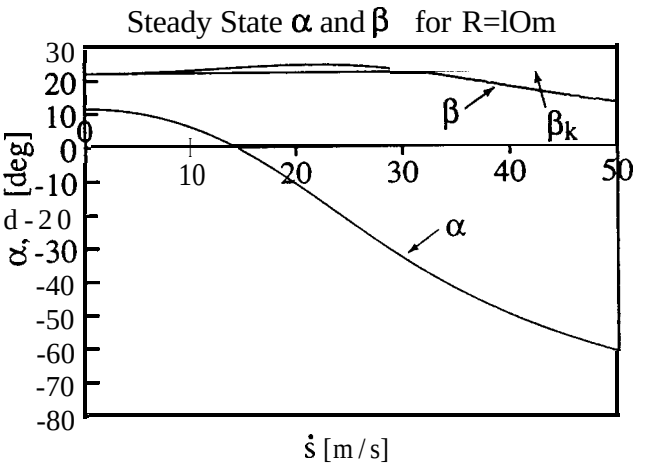

Figure 2.9: Steady-state α and β along a counter clockwise turn

ensures continuity of β, starting from known initial conditions ($\beta = 0$ along an initial straight line segment). Figure 2.10 also reveals that a solution for β might not exist for a too high or too low a_2. The trajectory must, therefore, satisfy at all $s \in [0, s_f]$ two constraints of the form

$$a_2 - a_{2max} \leq 0 \tag{2.48}$$

$$-a_2 + a_{2min} \leq 0 \tag{2.49}$$

where

$$a_{2max} = \max_{\beta \in [-\beta_{max}, \beta_{max}]} (a_1 - \beta)\cos(\beta) \tag{2.50}$$

$$a_{2min} = \min_{\beta \in [-\beta_{max}, \beta_{max}]} (a_1 - \beta)\cos(\beta) \tag{2.51}$$

in order to be dynamically feasible, and β_{max} represents the physical limit on the steering angle due to the steering mechanism:

$$|\beta| \leq \beta_{max}. \tag{2.52}$$

A trajectory segment might be infeasible if the acceleration component parallel to the y body axis, a_y, is too high, either because of a too high acceleration, speed, or path curvature.

This solution for β is valid only if the resulting $|\alpha_f| < \alpha_f^*$. Otherwise, we set the front lateral tire force, F_f, in (2.13) to its maximum value

$$F_f = F_f^* = -C_f \alpha_f^* sign(\alpha_f) \tag{2.53}$$

and solve for β, using:

$$\cos \beta = \frac{-1}{C_f \alpha_f^* sign(\alpha_f)} [F_r(\alpha) + m(\ddot{s} \sin \alpha + \dot{s}^2 K \cos a)] \tag{2.54}$$

Fwhere γ was defined in (2.4), and *sign(.)* is the signum function. Obviously, the right hand side of (2.54) should be in the interval [-1, 1] for a solution to exist. This in turn imposes a constraint on $\dot{s}$ and $\ddot{s}$, as discussed later. Alternatively, we can constrain β so that $|\alpha_f|$ does not exceed α_f^*.

Figure 2.9 shows the steady-state steering angle at various speeds along a counter clockwise turn of $10m$ radius. For this vehicle, the steering angle stays close to its kinematic value, $\beta_k = 22.1°$, until about $31m/s$, then it drops gradually to about 12" at $50m/s$. It is interesting to note that the constraints (2.48) and (2.49) were not active for the steady-state solution at any speed, whereas these constraints were active at relatively low speeds for

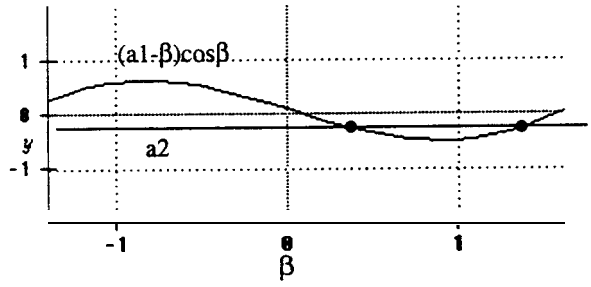

Figure 2.10: Typical curve of $y = (x - a)\cos(x)$ with $a = 0.1$.

the transient solutions. This suggests that the steady-state steering angle is a poor approximarion of the actual steering angle during extreme conditions.

Figures 2.11 and 2.12 show the steering angles for various constant speeds along the paths shown in figures 2.4 and 2.6, respectively. For the B-spline path (figure 2.11). one of the constraints (2.48) and (2.49) was violated for speeds exceeding $16m/s$, whereas for the circular turn (figure 2.12), this occurred above $20m/s$. The other constraint on β,(2.52), was not imposed in these computations.

Note the discontinuities and oscillations in β in figure 2.12 due to the discontinuities in path curvature, as compared to the smooth β at all speeds in figure 2.11. Because of the dynamics of the steering actuator, it would actually be impossible to follow the circular path at all but the lowest speeds. This strongly suggests that sharp transitions between straight and curved road segments are undesirable.

Once β has been computed for the entire path, the tractive force, F_t, is computed from the remaining equation of motion (2.12), subject to (2.19).

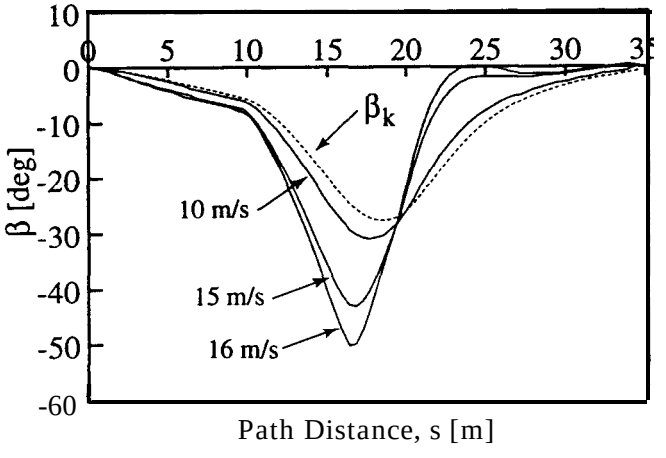

Figure 2.11: Dynamic β for various speeds along the B-spline path

Experience shows that it is generally difficult to plan a high speed velocity profile along a curved path without violating the steering angle constraint, mainly due to the oscillatory nature of α. It is therefore useful to transform the control constraints to the phase plane, s $-\dot{s}$, to produce upper bounds on vehicle speeds along the path, then plan dynamically feasible, and hence safe, trajectories in that plane, as discussed next.

2.4 Velocity and acceleration constraints

In planning emergency maneuvers, it is essential to avoid exceeding the tire force limits to prevent spin or skid. This can be done by mapping the tire force constraints to constraints on the vehicle's speed and acceleration, of the form

$$\ddot{s}_{min} \leq \ddot{s} \leq \ddot{s}_{max} \tag{2.55}$$

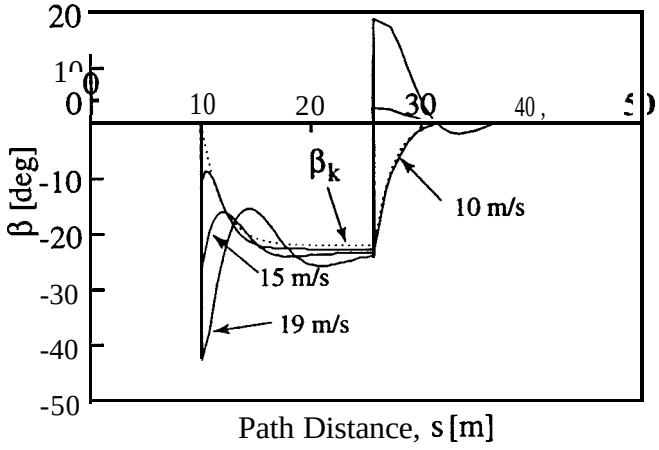

Figure 2.12: Dynamic β along the circular path

$$\dot{s} \leq \dot{s}_{max} \tag{2.56}$$

then using these bounds to verify that a given velocity profile along the path is indeed dynamically feasible.

The bounds on the acceleration, $\ddot{s}_{max}$ and $\ddot{s}_{min}$, are computed by, respectively, maximizing or minimizing $\ddot{s}$ over F_t and β, at any given (s, $\dot{s}, \alpha . \alpha_s$). Considering first the constraints on β, we substitute (2.47) into (2.48) and (2.49), to yield, respectively, for $\alpha > 0$:

$$\ddot{s} \geq \frac{1}{\sin\alpha}[-\dot{s}^2 K \cos\alpha - \frac{1}{m}(C_r\alpha_r + C_f a_{2max})] \tag{2.57}$$

$$\ddot{s} \leq \frac{1}{\sin\alpha}[-\dot{s}^2 K \cos\alpha - \frac{1}{m}(C_r\alpha_r + C_f a_{2min})] \tag{2.58}$$

Using the condition that the right hand side of (2.54) is within the interval [-1, 1] yields for $\alpha > 0$

$$\ddot{s} \leq \frac{1}{m\sin\alpha}[F_r(\alpha) - m\dot{s}^2 K \cos\alpha + C_f\alpha_f^*] \tag{2.59}$$

$$\ddot{s} \geq \frac{1}{m \sin \alpha}[F_r(\alpha) - m\dot{s}^2 K \cos \alpha - C_f \alpha_f^*] \tag{2.60}$$

and vice versa for $\alpha < 0$.

Now, assuming constant upper and lower limits on F_t due to the maximum engine torque and the maximum braking force

$$F_{tmin} \leq F_t \leq F_{tmax} \tag{2.61}$$

we obtain, considering the force equation (2.12),

$$\ddot{s} \leq \frac{1}{m \cos \alpha}[m\dot{s}^2 K \sin \alpha - F_f(\alpha) \sin \beta + F_{tmax}] \tag{2.62}$$

$$\ddot{s} \geq \frac{1}{m \cos \alpha}[m\dot{s}^2 K \sin \alpha - F_f(\alpha) \sin \beta + F_{tmin}] \tag{2.63}$$

Considering the friction ellipse (2.19), we obtain:

$$\ddot{s} \leq \frac{1}{m \cos \alpha}[A\sqrt{1 - (\frac{F_r}{B})^2} + C_f(\gamma - \beta) \sin \beta + m\dot{s}^2 K \sin \alpha] \tag{2.64}$$

$$\ddot{s} \geq \frac{1}{m \cos \alpha}[-A\sqrt{1 - (\frac{F_r}{B})^2} + C_f(\gamma - \beta) \sin \beta + m\dot{s}^2 K \sin \alpha] \tag{2.65}$$

Thus, $\ddot{s}_{max}$ is computed by evaluating (2.58) and (2.59), maximizing (2.62) and (2.64) over β, and selecting the lower of those values. The lower bound, $\ddot{s}_{\min}$ is computed similarly, using (2.57),(2.60),(2.63), and (2.65), and selecting the higher of those values. It is interesting to note that the bounds on $\ddot{s}$ are nonlinear in β, but linear in the bounds on F_t. Therefore, maximizing or minimizing $\ddot{s}$ is achieved by maximizing or minimizing F_t, respectively, if none of the constraints on β, F_f, and F_r are active.

We assumed that the upper bounds are higher than the lower bounds on $\ddot{s}$. This might not always be the case, giving rise to constraints on $\dot{s}$. The

upper bound on $\dot{s}$ can be determined by computing the lowest value of $\dot{s}$ for which the upper bound on $\ddot{s}$ for any one of the constraints is equal to the lower bound of the ***same*** constraint.

The bounds on the speed and acceleration apply locally to every point along the velocity profile. However, these bounds cannot guarantee that a trajectory, starting from a feasible initial state, is indeed feasible along the entire path. It is, therefore, useful to determine the set of initial conditions that yield feasible trajectories. This is done by computing the velocity limit curve, which serves as an upper bound on $\dot{s}$ at any given s, assuming $\alpha(0) = \alpha_s(0) = 0$.

2.5 Velocity Limit Curve

To compute the velocity limit curve along a specified path, we solve the following optimization problem, formulated in terms of path arc length, s, and its time derivatives. Denoting x = {s, $\dot{s}\}_T$, and $u = \ddot{s}$, we have:

$$\min_{u \in U} J = \int_0^{t_f} l dt \tag{2.66}$$

with free final time, subject to system dynamics

$$\dot{x}_1 = x_2 \tag{2.67}$$

$$\dot{x}_2 = u \tag{2.68}$$

subject to control constraints (2.55)

$$u_{min}(\mathbf{x}, \mathbf{y}) \leq u \leq u_{max}(\mathbf{x}, \mathbf{y}) \tag{2.69}$$

the boundary conditions

$$s(0) = s_0 , \; s(t_f) = s_f \tag{2.70}$$

$$y_1(0) = 0 \tag{2.71}$$

$$y_2(0) = 0 \tag{2.72}$$

where y = $\{\alpha, \dot{\alpha}\}$ are auxiliary variables. satisfying

$$\dot{y}_1 = y_2 \tag{2.73}$$

$$\dot{y}_2 = \frac{1}{I}(d_r C_r \alpha_r - d_f C_f(\gamma - \beta)\cos\beta) + \theta_s u + \theta_{ss} x_2^2 \tag{2.74}$$

and

$$\gamma = \tan^{-1}\left[\frac{d_f(\dot{\theta} - y_2) + x_2 \sin(y_1)}{x_2 \cos y_1}\right]$$

$$\alpha_r = \tan^{-1}\left[\frac{-d_r(\dot{\theta} - y_2) + x_2 \sin(y_1)}{x_2 \cos y_1}\right]$$

The boundary conditions on y assume that the vehicle is tangent to the path at the initial time. **Vehicle speeds** at the initial and final times, however, are unspecified to ensure that the fastest velocity profile is being selected.

2.6 Specified path optimization

The optimization of **the vehicle's** motion along a specified path is identical to the computation of the velocity profile, except that the initial speed is now specified. The optimal **velocity profile** is expected to stay below the velocity limit curve so as to satisfy **the tire** force and engine torque constraints.

A practical solution to this optimization problem is to represent the velocity profile along the path by a B-spline, then minimize the motion time, augmented with penalty functions on the control constraints. This transforms the optimization problem (2.66) to the following parameter optimization:

$$\min_{\mathbf{a}} J = \int_0^{t_f} 1 dt \qquad (2.75)$$

subject to the same constraints as problem (2.66), where a = $\{a_1, a_2, .., a,\}$ represents the variable control points of the velocity profile.

It is easy to show that the optimal control of this optimization problem is bang-bang, i.e. $\ddot{s}$ is either maximized or minimized at all times, since the Hamiltonian, H, is affine in the control:

$$H = 1 + \lambda_1 x_2 + \lambda_2 u \qquad (2.76)$$

where λ_1 and λ_2 are the two co-state variables. Following similar arguments as in (Shiller and Lu 1992), it can be shown that since, excluding singular arcs, $\lambda_2 \neq 0$, the Hamiltonian (3.12) is minimized only by maximizing or minimizing u $\equiv \ddot{s}$. This is similar to the structure of the optimal control of general robotic manipulators (Shiller and Lu 1992), except that here we have no systematic way to compute the switching points.

We developed an optimization routine, enhanced with interactive computer graphics, that computes the time optimal velocity profile along any given path. The path can be generated interactively by moving the control points on the screen, or generated from a data file of the control points. The initial velocity profile is also generated interactively, or from a data file. Once the velocity profile has been selected, the program computes the kinematic

Table 2.1: Parameters of the planar vehicle

d_f m	d_r m	m K	g I Kg-m^2	C_r N/rad	C_f N/rad
2	2	1,550	3,100	80,000	80,000

α_f^*	α_r^*	β_{max}	F_{tmax} N	F_{tmin} N	A N	B N
10”	10”	50”	3,000	-6,000	6,000	5,000

and dynamic steering along the path. The vehicle can then be moved interactively along the path for a visual comparison of the kinematic and dynamic solutions. An on-line dynamic simulation is also available for numerically verifying the correctness of the optimal solution. The evaluation of the vehicle's motions along a specified path is relatively fast, requiring only a few seconds. The computation of the velocity limit curve takes about an hour on an old Silicon Graphics 4D-70.

2.7 Example

In this example, we consider the path shown in figure 2.13, and the vehicle parameters given in table 1. The path consists of a cubic B-spline, represented by 8 control points. The curvature along this path is continuous, as shown in figure 2.14. The points of highest curvature are at s = 62m and $s = 79m$.

First, the velocity limit curve for this maneuver was computed by solving

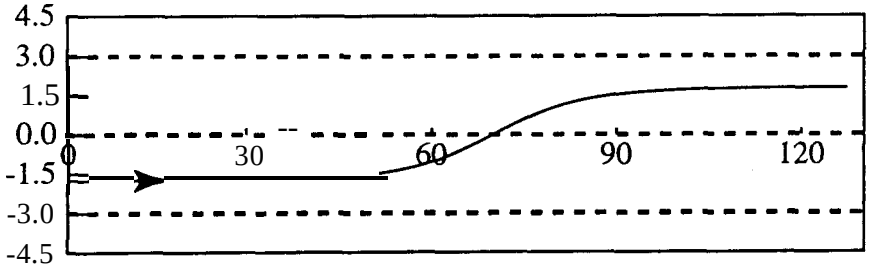

Figure 2.13: Path for lane change maneuver

the optimization problem (2.66), using a parameter optimization. Representing the velocity profile by a B-spline with 11 control points resulted in the velocity limit curve shown in figure 2.15, and in the corresponding tractive force shown in figure 2.16. The body slip angle and the steering angle are shown in figures 2.18 and 2.19, respectively. The transient steering angle deviates notably from the kinematic steering during the turns.

The velocity limit curve represents the highest speeds at which the vehicle can safely traverse this maneuver, with a motion time of 4.3s. The tangential acceleration, $\ddot{s}$, along the velocity limit curve is shown in figure 2.17. For the case shown, the vehicle slows down at the maximum deceleration of $3.9m/s^2$. It then switches gradually to the maximum acceleration, which reaches $1.9m/s$. The almost bang-bang structure of the tangential acceleration is expected, as concluded from (3.12). However, the switch from maximum deceleration to maximum acceleration is gradual due to the continuous representation of the velocity profile, which is C^2 continuous. It is evident from figure 2.16 that the constraints on the tractive force (2.61) are dominant along most of the path, except during the transition between the

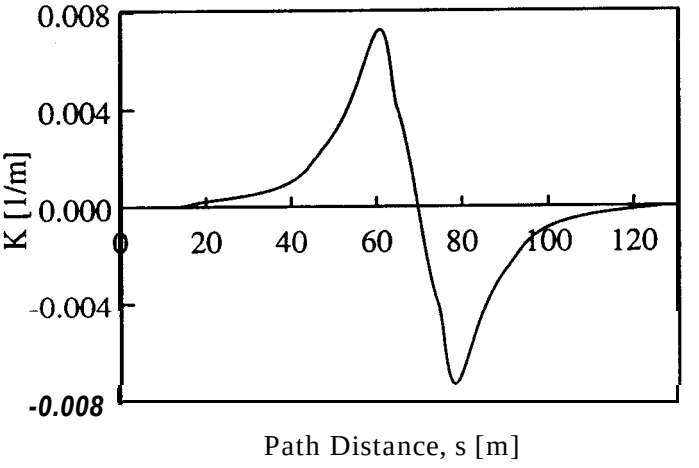

Figure 2.14: Curvature along the lane change maneuver

two peaks in the curvature where (2.22) is active at the points of highest curvature. No other constraint is active between these points due to the C^2 representation of the velocity profile.

Solving (2.66) with the additional constraint that the vehicle speed remains constant throughout the maneuver, resulted in the maximum constant speed of $27.9 m/s$, and a motion time of 4.7s from start to finish. For this case, the maximum constant speed touches the velocity limit curve at its lower point, as shown in figure 2.15. The active constraint in this case is the friction ellipse (2.22), which is active at s = 81m. The tire slip angles, shown in figure 2.20, are far from their critical values of 10^o, due to the conservative parameters chosen to represent the friction ellipse.

Interestingly, moving at a constant speed along this path does not require braking. The tractive force is positive during both left and right turns because the vehicle points inward during each turn, as seen from the slip angle

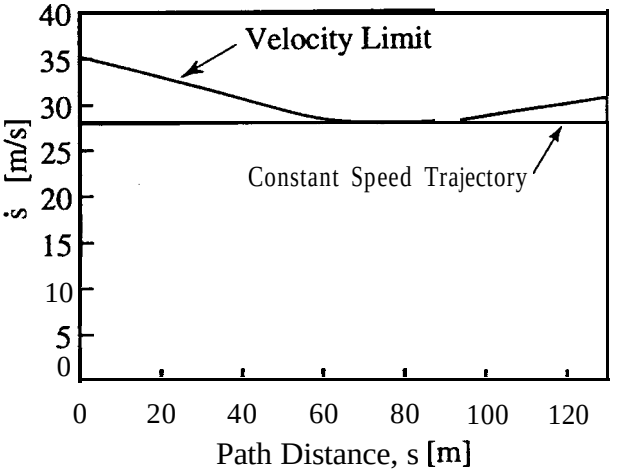

Figure 2.15: Velocity profiles for the lane change maneuver

shown in figure 2.18. Hence, a positive tractive force would have a positive component in the direction of the centripetal acceleration, and thus aid in traversing the turn.

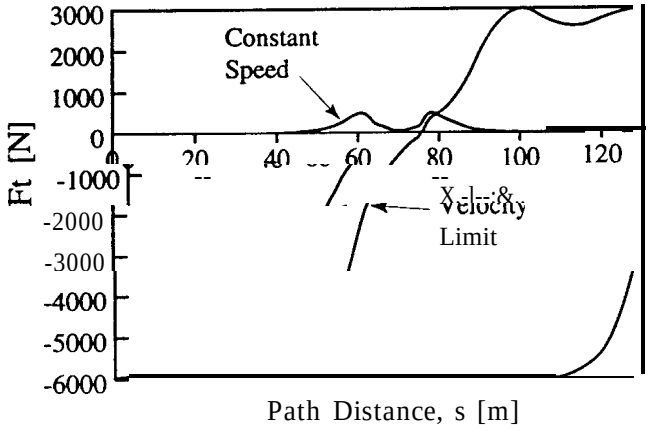

Figure 2.16: Tractive force for velocity limit and maximum constant speed along lane change maneuver

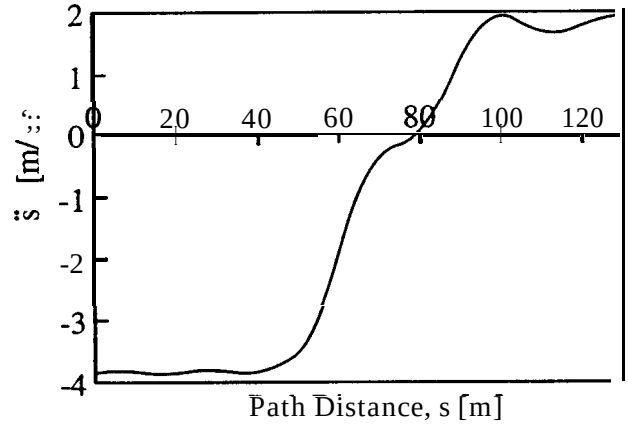

Figure 2.17: Acceleration limit for lane change maneuver

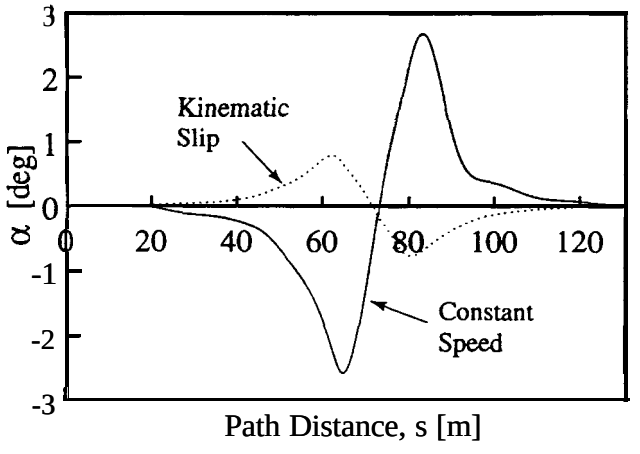

Figure 2.18: Slip angle for maximum constant speed along the lane change maneuver

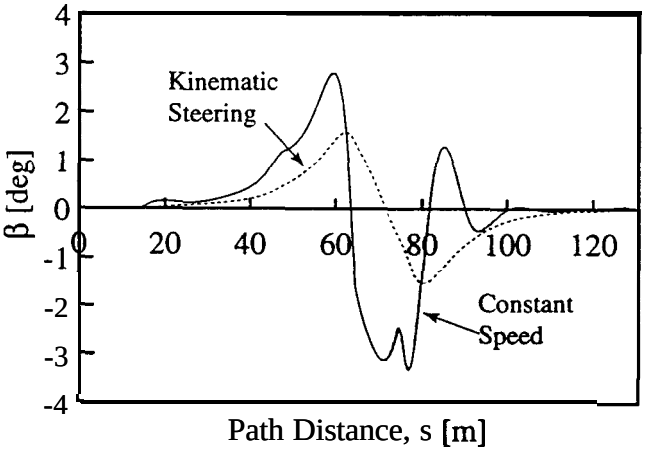

Figure 2.19: Steering angle for maximum constant speed along the lane change maneuver

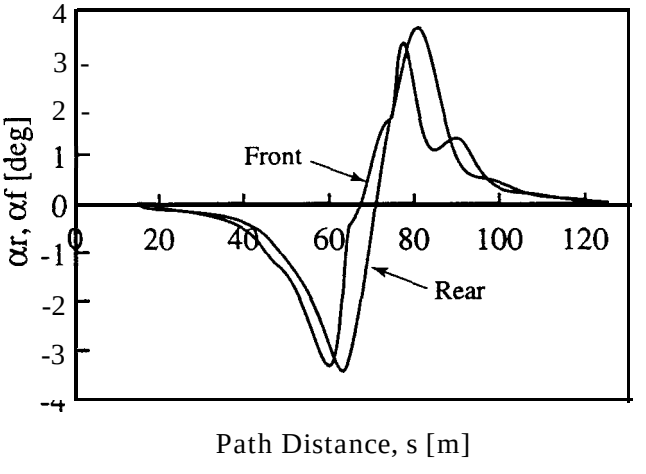

Figure 2.20: Tire slip angles for maximum constant speed along the lane change maneuver

Chapter 3

Optimal Lane-Change Maneuvers

This chapter presents the computation of optimal lane-change maneuvers for collision avoidance. We use the same bicycle model as before, except that this model is not transformed to path coordinates (arc length and speed) since the optimization problem here is formulated as an unconstrained parameter optimization between given and free boundary conditions, with no relation to the specified path problem.

3.1 Emergency Lane-Change Maneuver

A typical emergency lane-change maneuver is shown in figure 3.1, where a vehicle, moving at some initial speed, $\dot{x}_0$, avoids an obstacle that is blocking its forward path. In the context of this project, the optimal maneuver for

collision avoidance is one that avoids the obstacle from the closest distance, or, equivalently, that minimizes the distance traveled in the current lane. This maneuver is computed by minimizing the longitudinal displacement of the entire lane transition, subject to vehicle dynamics and constraints on the steering angle and the tire forces, as is formulated in the following optimization problem.

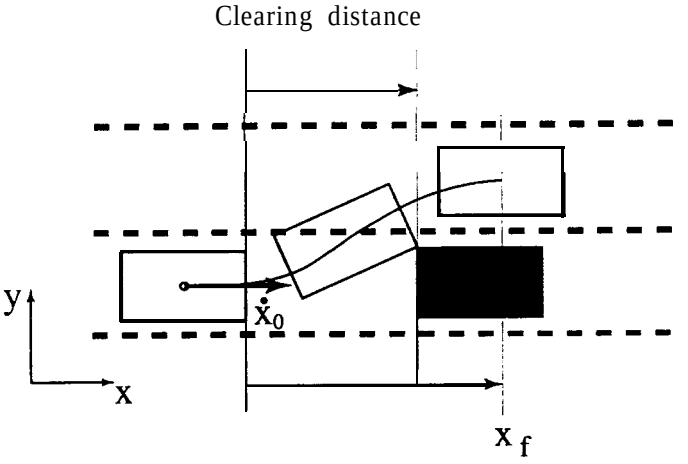

Figure 3.1: Lane-change maneuver

3.1.1 Problem Formulation

Denoting x $=\{x, \dot{x}, y, \dot{y}, \varphi, \dot{\varphi}\}^T$, and u = $\{F_t, \beta\}^T$, we solve the following optimization problem:

$$\min_{\mathbf{u}} J = x_1(t_f) = \int_0^{t_f} x_2(\mathbf{x}, \mathbf{u}, t)dt \tag{3.1}$$

with free final time, t_f, subject to system dynamics (2.9)

$$\dot{x}_1 = x_2 \tag{3.2}$$

$$\dot{x}_2 = \frac{1}{m}(u_1 \cos x_5 - F_r \sin x5 - F_f \sin(x_5 + u_2)) \quad (3.3)$$

$$\dot{x}_3 = x_4 \quad (3.4)$$

$$\dot{x}_4 = \frac{1}{m}(F_r \cos x_5 + u_1 \sin x_5 + F_f \cos(x_5 + u_2)) \quad (3.5)$$

$$\dot{x}_5 = x_6 \quad (3.6)$$

$$\dot{x}_6 = \frac{1}{I}(-d_r F_r + d_f F_f \cos u_2) \quad (3.7)$$

the boundary conditions

$$x_1(0) = 0$$

$$x_2(0) = \dot{x}_0$$

$$x_3(t_f) = y_d$$

$$x_4(0) = x_4(t_f) = 0$$

$$x_5(0) = x_6(0) = x_5(t_f) = x\&f) = 0$$

and the control constraints

$$g_1(\mathbf{u}) = u_2 - \beta_{max} \leq 0 \quad (3.8)$$

$$g_2(\mathbf{u}) = \beta_{min} - u_2 \leq 0 \quad (3.9)$$

$$g_3(\mathbf{x}, \mathbf{u}) = (\frac{u_1}{F_{xmax}})^2 + (\frac{F_r}{F_{ymax}})^2 - 1 \leq 0 \quad (3.10)$$

$$g_4(u_1) = u_1 - u_{1max} \leq 0 \quad (3.11)$$

where y_d is the distance between the centers of adjacent lanes, and, using (2.5), (2.6) and (2.21)

$$F_f = -C_f(tan^{-1}[\frac{d_f x_6 + x4 \cos x5 - x_2 \sin x5}{x_2 \cos x_5 + x_4 \sin x_5} \text{I-} u_2)$$

$$F_r = -C_r tan^{-1}[\frac{-d_r x_6 + x_4 \cos x_5 - x_2 \sin x_5}{x_2 \cos x_5 + x_4 \sin x_5}].$$

The constraint (3.11) is due to the maximum tractive force, u_{1max}, generated by the maximum engine torque. This constraint applies only to a positive force since the negative force, generated by the brakes, is bounded by the friction ellipse (3.10).

3.1.2 Structure of Optimal Control

It is easy to show that the optimal control for this problem is bang-bang in F_t since system dynamics, and hence the Hamiltonian, H, are affine in $u_1 \equiv F_t$:

$$H = x_2 + \lambda_1 x_2 + \lambda_3 x_4 + \lambda_5 x_6 + \left(\frac{\lambda_2}{m} \cos x_5 + \frac{\lambda_3}{m} \sin x_5 \right) u_1 + h(\lambda, \mathbf{x}, u_2) \tag{3.12}$$

where $\lambda_i, i = 1, \ldots, 6$, are the co-state variables, and $h(\lambda, \mathrm{x}, u_2)$ is a nonlinear function of x and u_2. Excluding singular arcs, the Hamiltonian (3.12) is minimized by either maximizing or minimizing u_1. The optimal u_2 (the steering angle) is calculated from $H_{u_2} = 0$. Note that this is also the structure of the equivalent time optimal control problem.

3.1.3 Trajectory Optimization

The optimization problem (3.1) is reformulated as a parameter optimization. Representing the trajectory, x(t), by a finite set of parameters, a = $\{a_1, a_2, \ldots, \mathrm{a},\}$, transforms (3.1) to

$$\min_{\mathbf{a}} J = x_1(t_f(\mathbf{a})) \tag{3.13}$$

subject to the same constraints as (3.1). 0 nce the optimal trajectory has been found, the optimal controls are computed using inverse dynamics (Section 2.3).

For the problem treated here, the path is represented by the control points of a cubic B-spline, and the velocity profile along the path is represented by a set of discrete points. The control constraints are appended to the cost function using penalty functions.

Problem (3.13) can be solved using standard gradient methods. The gradient, though, needs to be computed numerically since the cost function is not explicit in the optimization parameters.

Formulating the optimization problem as a parameter optimization over parameters associated with the trajectory, rather than the controls, was recently introduced by Seywald (1994) as Differential Inclusions, and by Bryson (1995) as Inverse Dynamic Optimization. A similar approach was previously used for optimizing robot motions (Shiller and Dubowsky, 1989).

This optimization scheme is computationally more efficient than traditional variational methods since it does not require co-states, state constraints can be easily considered (transformed to constraints on the parameters), and boundary conditions are easily satisfied. In addition, this optimization scheme can be terminated (if computation time is bounded) to produce sub-optimal but feasible solutions, which is not the case with variational methods that must satisfy necessary optimality conditions (Bryson and Ho, 1975).

3.2 Minimum Clearing Distance

In addition to being used as reference trajectories for on-line feedback controllers, the optimal maneuvers obtained by solving (3.1) can be used to determine the minimum cleating distance, which is the distance beyond which an obstacle cannot be avoided for given initial speeds.

The minimum *clearing distance,* $x_c(\dot{x}_0)$, is defined as:

$$x_c(\dot{x}_0) = \min x \tag{3.14}$$

where x is the distance between the vehicle and obstacle, subject to the condition

$$A(t, \dot{x}_0) \cap \mathrm{B(x)} = 0,\ t \in [0, t_f], x > 0 \tag{3.15}$$

where $A(t, \dot{x}_0)$ represents the volume swept by the moving vehicle along the optimal trajectory at time t, and B(x) represents the volume occupied by the static obstacle, positioned in the current lane at a distance x from the vehicle.

The minimum cleating distance can be computed by determining the point at which a vertex of the vehicle moving along the optimal trajectory intersects a vertex of the static obstacle, as shown in figure 3.2.

For a left lane transition, the points of concern are the front right corner of the moving vehicle and the rear left corner of the obstacle. Let the optimal trajectory be $\mathbf{x}(t, \dot{x}_0) = \{x(t, \dot{x}_0), y(t, \dot{x}_0), \varphi(t, \dot{x}_0)\}$. Then, the time, t_c, at which the two vertices coinside is the time when the y coordinate of the front right corner of the vehicle passes through y = b/2 (assuming the vehicle and

obstacle are of identical geometries):

$$y(t_c, \dot{x}_0) + d_f \sin \varphi(t_c, \dot{x}_0) - \frac{b}{2} \cos \varphi(t_c, \dot{x}_0) = \frac{b}{2} \tag{3.16}$$

The minimum *clearing distance,* $x_c(\dot{x}_0)$, is then

$$x_c(\dot{x}_0) = x(t_c, \dot{x}_0) + d_f \cos \varphi(t_c) + \frac{b}{2} \sin \varphi(t_c) \tag{3.17}$$

where b is the vehicle's width.

Plotting $\dot{x}_0$ cs. $x_c(\dot{x}_0)$ for the optimal maneuvers, computed for various initial speeds, produces a curve in the phase plane $\dot{x} -$ x, as shown schematically in figure 3.3. This curve marks the boundary of the vehicle's states (position and speed) from which an obstacle can be avoided. States right of the curve are avoidable, whereas states left of the curve are not.

It is useful to add the *stopping distance* curve, representing states in the phase-plane from which the vehicle can decelerate to a full stop and just touch the obstacle, using the maximum deceleration, as shown schematically in figure 3.3. The stopping *distance* is simply computed by:

$$\dot{x} = \sqrt{2\ddot{x}_{min} x} \tag{3.18}$$

where $\ddot{x}_{min}$ is the maximum longitudinal deceleration, which depends on the vehicle weight and road conditions.

These curves can assist in planning an avoidance maneuver. For example, detecting the obstacle at a speed and distance corresponding to Region I, leaves sufficient time for a full stop, or for a relaxed lane transition. Detecting the obstacle in Region II (for example, point a) leaves a lane transition or a head-on collision as the only options. An avoidance maneuver in Region II

may consist of a relaxed[1] maneuver from point a. Alternatively, the avoidance maneuver may consist of a deceleration, until hitting the *clearing distance* curve, then an optimal maneuver corresponding to a lower speed (point *b).* The preferred maneuver depends on the congestion in the neighboring lanes. The deceleration-and-optimal maneuver would allow the longest time in the current lane. Crossing the clearing *distance* curve into Region III (for example, point c) eliminates the lane-change option. The best strategy then is to stay in the current lane, decelerate at the maximum deceleration, and brace for a head-on collision at a lower speed (point *d).* Attempting a lane transition from points in Region III may result either in an off-center collision, or loss of control. Either is generally more dangerous than a head-on collision, for which the vehicle is better designed to sustain.

In addition to assisting in planning collision-avoidance maneuvers, the *clearing distance* curve can be used to specify the minimum range of collision-avoidance sensors. Clearly, any collision-avoidance sensor must have a minimum range outside of Region III.

3.3 On-line approximation of emergency maneuvers

The optimization of a lane change maneuver is computationally too expensive for on-line applications. One approach to overcome the computational

[1] In the contest of this work, a relaxed maneuver is one that satisfies criteria not essential for safety, such as ride comfort

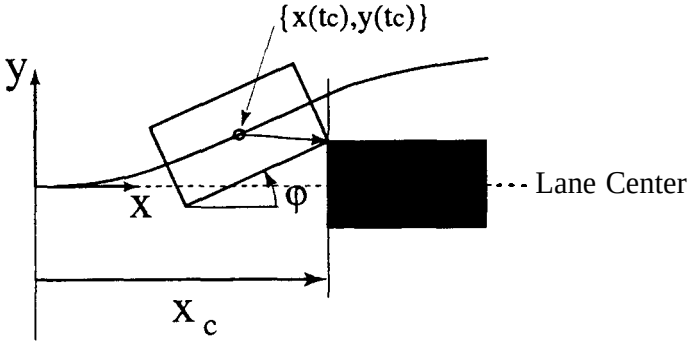

Figure 3.2: Calculating the minimum clearing *distance*

burden is to use a table look-up of precomputed optimal maneuvers, tabulated for various initial speeds and road conditions. To reduce the resolution of the table look-up with respect to the initial speed, we propose to extrapolate maneuvers from a nominal optimal maneuver. Another approach is to approximate the optimal maneuvers using a simple point mass model. This may not produce an optimal maneuver, but it can account for the actual initial speed, vehicle parameters, and road conditions. In the following, we first discuss the point mass model, and then present the extrapolation method, which is based on insights gained from the point mass model.

3.3.1 The Point Mass Model

The point mass model consists of a point mass, forced by two independent perpendicular forces, as shown in figure 3.4. The optimal lane-change ma-

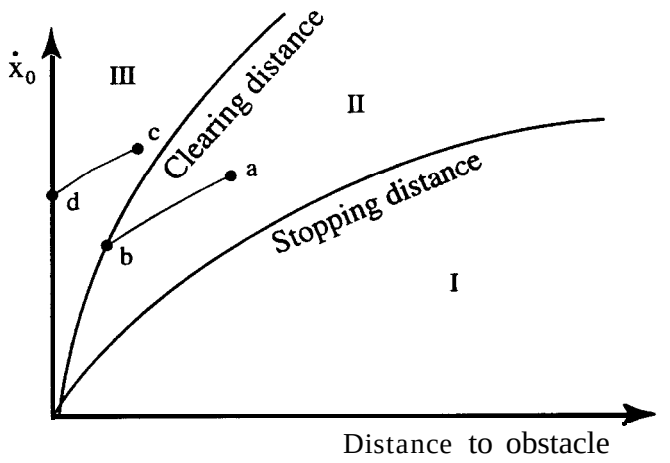

Figure 3.3: The minimum clearing *distance* in the phase-plane

neuver for this system is computed by solving the following optimization problem

$$\min J = x(t_f) \tag{3.19}$$

where t_f is free, subject to system dynamics

$$\ddot{x}(t) = \frac{F_x}{m} \tag{3.20}$$

$$\ddot{y}(t) = \frac{F_y}{m} \tag{3.21}$$

the control constraints

$$|F_x| \leq F_{xmax} \tag{3.22}$$

$$|F_y| \leq F_{ymax} \tag{3.23}$$

and boundary conditions

$$\dot{x}(0) = \dot{x}_0 \tag{3.24}$$

$$\dot{x} = \text{free} \tag{3.25}$$

$$y(0) = x(0) = 0, \tag{3.26}$$

$$y(t_f) = y_d, \tag{3.27}$$

$$\dot{y}(0) = \dot{y}(t_f) = 0 \tag{3.28}$$

where y_d is the distance between the centers of adjacent lanes.

Problem (3.19) consists of two separate problems that are coupled by the boundary conditions. The first problem is to minimize the motion distance in the x direction for $t \in [0, t_f]$. The final time, t_f, is determined by the motion in the y direction. Since x_f is minimized if t_f is minimized (assuming $\dot{x} > 0$), the motion in the y direction should be of minimum time. The optimal control in the y direction is, therefore, bang-bang with one switch at $t = \frac{t_f}{2}$. The motion in the x direction minimizes x_f, which is equivalent to minimizing $\dot{x}$:

$$\min\ J = x(t_f) = \int_0^{t_f} \dot{x} dt \tag{3.29}$$

The optimal control for problem (3.29) is thus the maximum braking force.

Integrating the optimal control twice with respect to time yields analytical expressions for $x(t)$ and $y(t)$ as functions of the initial speed $\dot{x}_0$, the vehicle mass, m, and the actuator constraints:

$$x(t) = \dot{x}_0 t - \frac{1}{2}\frac{F_{xmax}}{m} t^2 \tag{3.30}$$

$$y(t) = \begin{cases} \frac{1}{2}\frac{F_{ymax}}{m} t^2, & 0 < t < \frac{t_f}{2} \\ 2\sqrt{y_d \frac{F_{ymax}}{m}}\, t - \frac{1}{2}\frac{F_{xmax}}{m} t^2 - y_d, & \frac{t_f}{2} < t \leq t_f \end{cases} \tag{3.31}$$

where

$$t_f = 2\sqrt{\frac{m y_d}{F_{ymax}}} \tag{3.32}$$

$$x_f = \dot{x}_0 t_f - \frac{1}{2}\frac{F_{xmax}}{m} t_f^2 \qquad (3.33)$$

It is assumed that $\dot{x}(t) > 0$ for all $t \in [0, t_f]$. To ensure forward motion, we may restrict the initial speed or reduce F_{xmax} so that

$$\dot{x}_0 > \frac{F_{xmax}}{m} t_f \qquad (3.34)$$

The optimal maneuvers for the point mass model can be easily computed using (3.30) and (3.31) for any given initial speed. The actuator limits, F_{xmax} and F_{ymax}, can be chosen conservatively to ensure that the resulting maneuver is dynamically feasible. The smaller the force limit, the longer the maneuver. This optimization scheme can be used to produce maneuvers with specified terminal times, or be modified to maximize ride comfort by adding jerk constraints.

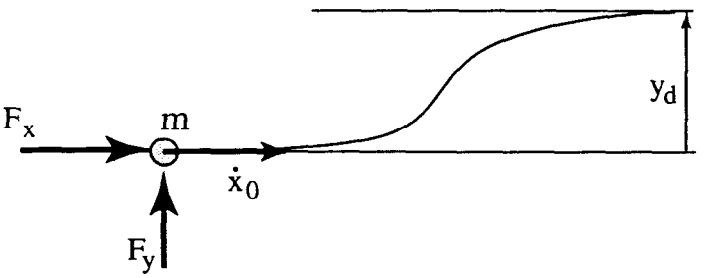

Figure *3.4:* The point mass model

3.3.2 Extrapolating from a Nominal Maneuver

The insights gained for the point mass model can be used to approximate maneuvers from an optimal maneuver computed for a nominal initial speed. This is based on the observation that the motion in the x direction for the point mass model is linear in $\dot{x}(0)$. The final time, t_f, is determined by the motion in the y direction, independent of $\dot{x}(0)$. It follows that the optimal trajectory $\mathbf{x}_0(\dot{x}_1(0), t)$ (for the point mass model), corresponding to the initial speed $\dot{x}_1(0)$, can be extrapolated from the optimal trajectory $\mathbf{x}_0(t) = \{x_0(t), y_0(t)\}$ corresponding to the initial speed $\dot{x}_0(0)$:

$$y_1(t) = y_0(t) \tag{3.35}$$

$$x_1(\dot{x}_1(0), t) = x_0(t) + (\dot{x}_1(0) - \dot{x}_0(0))t \tag{3.36}$$

Thus, the optimal trajectory for the point mass model at any initial speed can be easily extrapolated, using (3.35) and (3.36), from a single optimal trajectory.

To verify that this extrapolation is indeed valid for the bicycle model, we use (3.36) to define the variable $z(t)$:

$$z(t) = \frac{x_1(\dot{x}_1, t) - x_0(t)}{\dot{x}_1(0) - \dot{x}_0(0)}, t \in [0, t_f] \tag{3.37}$$

For the point mass model, $z(t)$ is linear in t ($z(t) = t$). The linearity of $z(t)$ for a specific family of maneuvers can be used as a quality measure of the extrapolated maneuvers, computed using (3.35) and (3.36), as demonstrated in the following examples.

3.4 Examples

The optimization of the lane-change maneuvers was implemented on a Silicon Graphics workstation for the planar bicycle model, using the parameters given in Table 1.

3.4.1 Optimal Maneuvers

Using 12 control points to represent the path, and 7 points to represent the velocity profile, resulted in the optimal maneuvers shown in figure 3.5. As is evident from figure 3.5, the higher the speed, the longer the maneuver. The total motion time for these maneuvers, though, changed little with the initial speeds (2.6s at $20m/s$ to 2.3s at $50m/s$). The optimal velocity profiles for these maneuvers are shown in figure 3.6. The active constraint in all optimal maneuvers was the friction ellipse (3.10).

Figure 3.i shows the optimal steering angles along three maneuvers, normalized with respect to the total path distance. Generally, the higher the speeds, the smaller and more oscillatory the steering angle.

Figure 3.S shows the actual cleating *distance* curve for the optimal maneuvers shown in figure 3.5. It is roughly a straight line (the slight deviations may be due to small numerical errors). This line was computed for maneuvers from $50m/s$ down to $10m/s$. It was extrapolated for lower speeds, as shown by the dashed line in figure 3.8. The slope of this line for this case is almost Is-^{1}, however, this slope depends on the vehicle and road parameters.

The clearing *distance* curve is significantly different from the hyperbolic *stopping distance* curve shown in figure 3.8 for $\ddot{x}_{min} = 3.87m/s^2$. The dif-

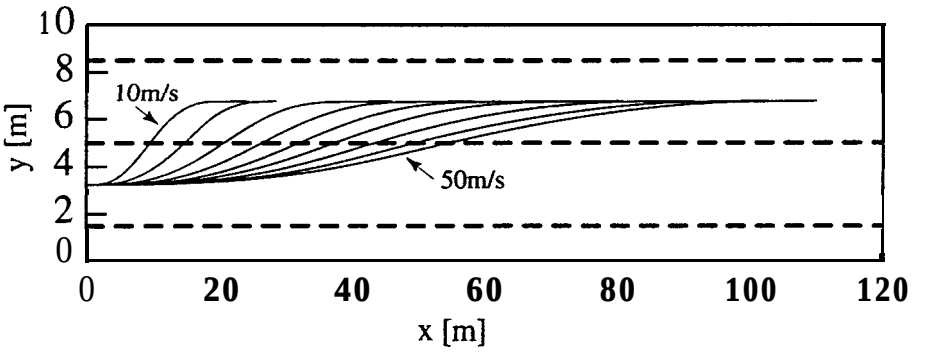

Figure 3.5: Optimal lane change maneuvers

ference between the two curves clearly demonstrates the advantage of the optimal maneuvers over a full stop at high speeds. For example, a vehicle traveling at $30m/s(108Km/h)$ requires 116m for a full stop, whereas it requires only 30m for an optimal lane-change maneuver.

The time in the current lane may be maximized by decelerating as soon as the obstacle is detected, until reaching the clearing *distance* curve at a lower speed. For example, detecting the obstacle from a distance of 80m at $30m/s$ (leaving no option for a full stop) and maintaining the current speed would allow 1.6s before the vehicle must execute an optimal maneuver. Decelerating at the maximum deceleration would extend the time in the current lane to 2.3s before reaching the *clearing distance* at $21m/s$, an increase of 40%. The extra time can be used for communication with the neighboring vehicles and coordination of the lane transition.

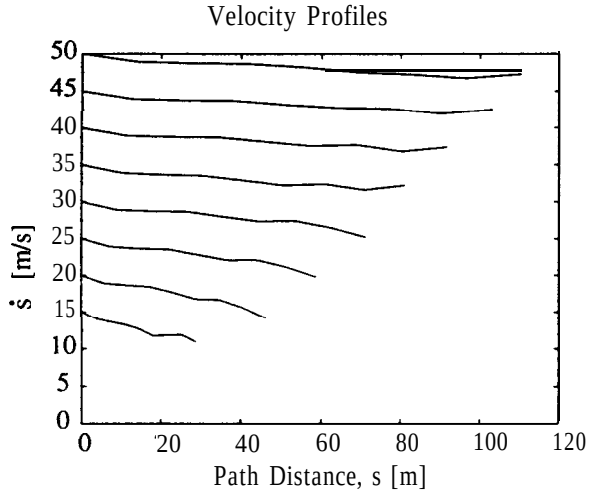

Figure 3.6: Velocity profiles along optimal lane change maneuvers

Table 3.1: Parameters of the planar vehicle

d_f [m]	d_r [m]	m [Kg]	I [Kg-m^2]	C_r [N / r a d]	C_f [N/rad]
2	2	1,550	3,100	80,000	80,000

3.4.2 Approximated Maneuvers-Point Mass Model

Computing the optimal maneuvers for the point mass model, using the major axes of the friction ellipse as the force limits, resulted in the *approximated* maneuvers shown in figure 3.9. These maneuvers are remarkably close to the optimal maneuvers computed for the nonlinear bicycle model. The velocity profiles of the point mass model are steeper (higher deceleration) than those for the bicycle model, as can be seen in figure 3.10. It is interesting to compare

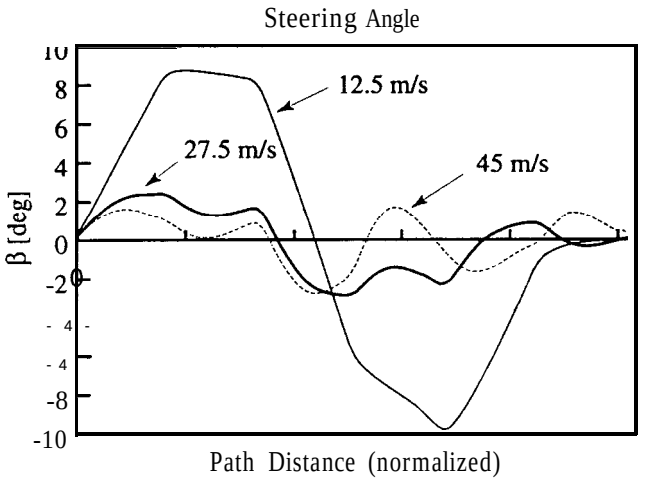

Figure 3.7: Steering angle along optimal maneuvers

α^*	β_{max}	F_{tmax} [N]	F_{tmin} [N]	F_{xmax} [N]	F_{ymax} [N]
10”	50”	*3,000*	*-6,000*	*6,000*	*5,000*

the total motion times for the optimal and the approximated maneuvers. For the approximated maneuvers, it is constant, at $2.08s$, independently of the initial speed, whereas for the optimal maneuver it varies little from 2.3 for $50m/s$ to 2.6s for $20m/s$. The optimal maneuvers are slightly longer due to the lower deceleration during the turns. This suggests that the point mass model can be used to closely approximate the paths of the lane change maneuvers. The velocity profile along the path can be computed using the method presented in (Shiller and Sundar, 1995), which is generally simpler

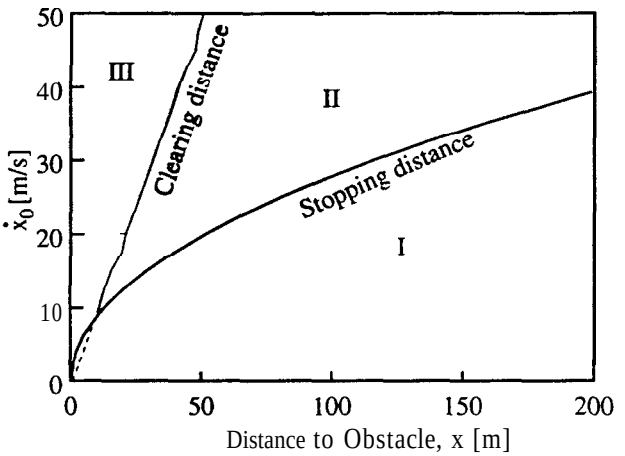

Figure 3.8: Minimum cleating distance for the optimal maneuvers of figure 5

and computationally more efficient than the optimization of the entire lane-change maneuver.

3.4.3 Extrapolated Maneuvers

Computing (3.37) for the optimal maneuvers of the bicycle model (figure 3.5), using the maneuver for $\dot{x}(0) = 45m/s$ as the nominal trajectory, resulted in the curves shown in figure 3.11. These curves deviate only slightly from the straight line of the point mass model, suggesting the validity of the extrapolation scheme.

Figure 3.12 shows the maneuvers extrapolated, using (3.35) and (3.36), from the optimal trajectory for $\dot{x}(0) = 45m/s$. They are remarkably close to the optimal maneuvers, as are their velocity profiles, shown in figure 3.13. The closer the initial speed to $45m/s$, the better the approximation. This

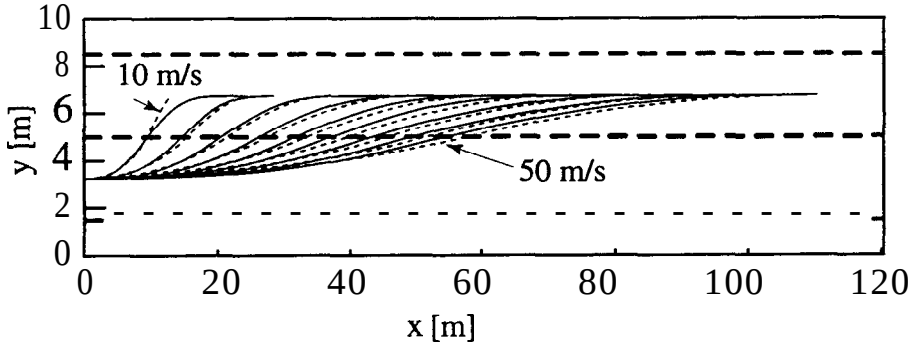

Figure 3.9: Optimal (solid) and approximated (dashed) maneuvers

confirms that the optimal maneuvers can indeed be extrapolated on-line from a few optimal maneuvers: optimized for low, moderate, or high speeds.

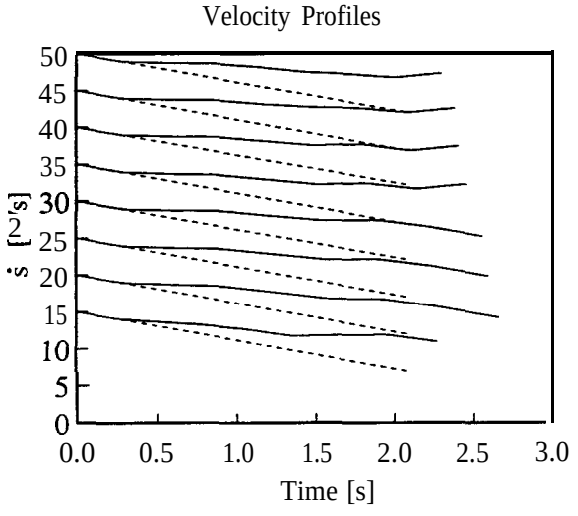

Figure 3.10: Velocity profiles of optimal (solid) and approximated (dashed) maneuvers

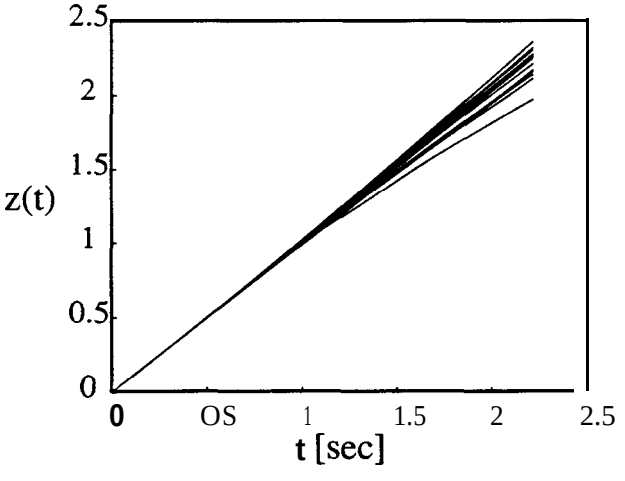

Figure 3.11: Difference between optimal and nominal maneuvers

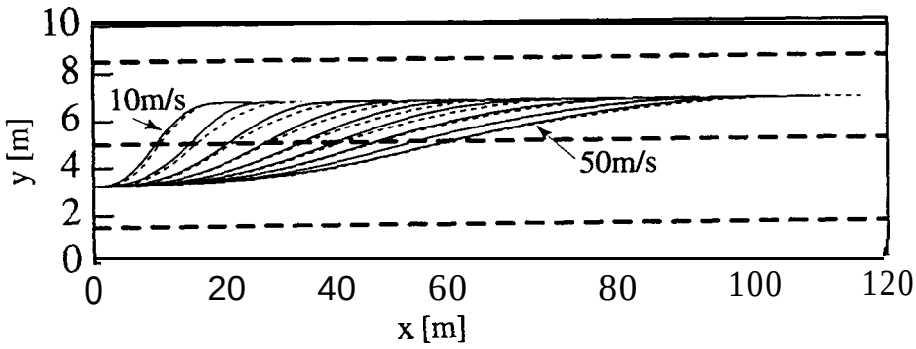

Figure 3.12: Optimal (solid) and extrapolated (dashed) maneuvers

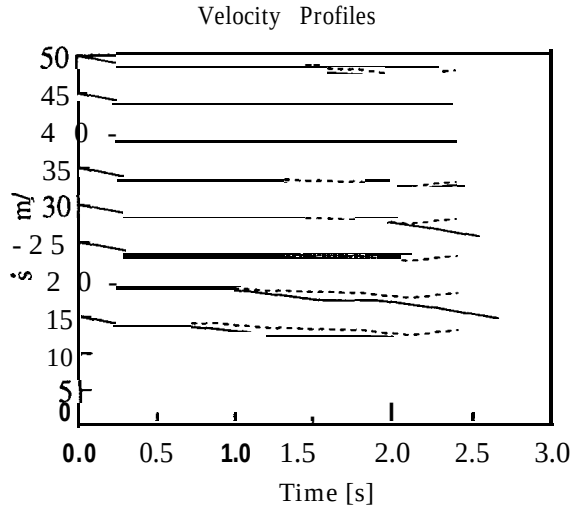

Figure 3.13: Optimal (solid) and extrapolated (dashed) velocity profiles

Chapter 4

6D Dynamic Model

Here we derive the equations of motion of a vehicle, modeled as a rigid body moving on a horizontal plane, as shown in figure 4.1. The vehicle is suspended on four springs, which are connected four wheels. The springs represent the suspension and tire compliance. The forces are generated between the tires and ground, transfered without losses (assuming massless and frictionless wheels) to the wheel axis.

We define an inertial frame O, and a body frame C, as shown in figure 4.1. The generalized coordiantes are chosen as $\{x, y, z\}$ in the inretial frame, and the three Euler angles $\{\varphi, \psi, \phi\}$, corresponding to first rotation around the z axis, producing the frame A , then rotation around the y axis, producing frame B, and finally rotation around the x axis, producing the frame C, as shown in figure 4.2.

Deriving the equations of motion using Lagrange's formulation, we first

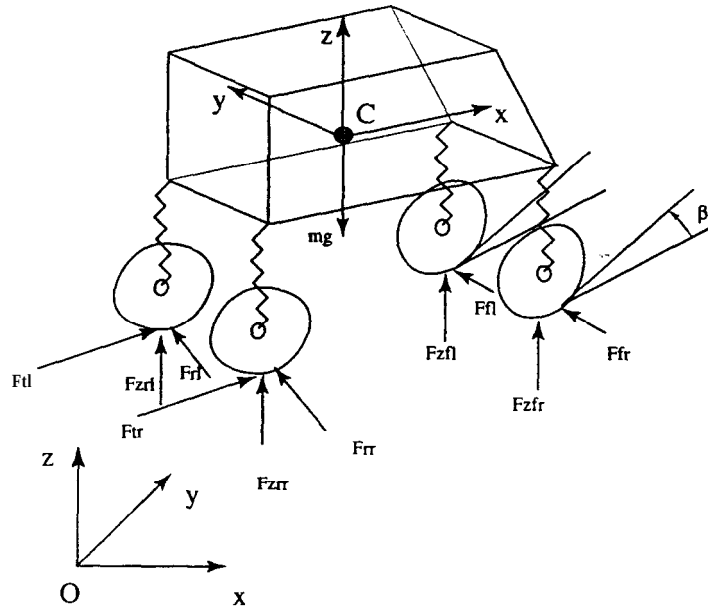

Figure 4.1: The vehicle and external forces

derive the total kinetic energy of the moving body in the form

$$E_k = \frac{1}{2}\dot{q}^T M(q)\dot{q} \tag{4.1}$$

where $M(q)$ is the equivalent inertia matrix, q is the vector of the generalized coordiantes, and $\dot{q}$ is the vector of the generalized velocities (time derivatives of the generalized coordiantes).

Denoting the potential energy E_p, the equations of motion can be derived

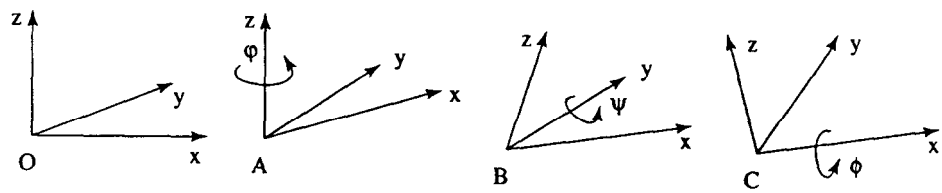

Figure 4.2: Rotation axes and rotated coordiante frames

directly from $M(q)$ and E_p, using:

$$\mathtt{Q = M(q)@ + } c(q,\dot{q}) \mathtt{ - } \frac{\partial E_p}{\partial q} \tag{4.2}$$

where Q is the vector of generlized forces, and $c(q,\dot{q})$ is the vector of centrifugal and Coriolis forces, with a typical element $c_i(q,\dot{q})$:

$$c_i(q,\dot{q}) = \dot{q}^T C_i(q)\dot{q} \tag{4.3}$$

where

$$C_i(q) = \frac{\partial M_i(q)}{\partial q} - \frac{1}{2}\frac{\partial M(q)}{\partial q_i} \tag{4.4}$$

where $M_i(q)$ is the ith row of $M(q)$.

4.1 Kinetic Energy

The kinetic energy of a moving rigid body is

$$E_k = \frac{1}{2}m(\dot{x}^2 + \dot{y}^2 + \dot{z}^2) + \frac{1}{2}\omega_c^T I_c \omega_c \tag{4.5}$$

where ω_c is the absolute rotation, expressed in the body frame (frame C in figure ??), and I_c is the moment of inertia in the body frame.

We now rewrite the rotational part of the kinetic energy in the form (4.1). The principal moments of inertia are assumed to coincide with the body frame, hence:

$$I_c = \begin{vmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{vmatrix} \tag{4.6}$$

The absolute angular velocity is the sum of the angular velocities around each axis:

$$\omega_c = \omega_\varphi + \omega_\psi + \omega_\phi \tag{4.7}$$

where the angular velocity around an individual axis is denoted by the rotation angle. The angular velocity around each individual axis can be derived in terms of the generalized coordinates, and projected to the body frame. Starting with the rotation around the x axis:

$$\omega_x = \begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix} \dot{\phi} \tag{4.8}$$

The rotation around the y axis, projected to the body frame, is:

$$\omega_y = T_C^B(\phi) \begin{bmatrix} 0 \\ \dot{\psi} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} 0 \\ \dot{\psi} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \cos(\phi) \\ -\sin(\phi) \end{bmatrix} \dot{\psi} \tag{4.9}$$

where T_C^B denotes the rotation matrix from frame B to frame C.

The rotation around the z axis, projected to the body frame, is:

$$\omega_z = T_C^A(\phi, \psi) = \begin{bmatrix} -\sin(\psi) \\ \cos(\psi)\sin(\phi) \\ \cos(\psi)\cos(\phi) \end{bmatrix} \dot{\varphi} \tag{4.10}$$

where T_C^A is the rotation matrix from frame A to frame C:

$$T_C^A(\phi, \psi) = T_C^B(\phi) T_B^A(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & \sin(\phi) \\ 0 & -\sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} \cos(\psi) & 0 & -\sin(\psi) \\ 0 & 1 & 0 \\ \sin(\psi) & 0 & \cos(\psi) \end{bmatrix} =$$

$$\begin{bmatrix} \cos(\psi) & 0 & -\sin(\psi) \\ \sin(a)\ \sin(\psi) & \cos(\phi)\ \sin(\phi)\ \cos(\psi) \\ \cos(\phi)\ \sin(\psi) & -\sin(\phi)\ \cos(\phi)\ \cos(\psi) \end{bmatrix} \tag{4.11}$$

Summing the angular velocities due to the rotations around the individual axes yields the absolute angular velocity expressed in the body frame:

$$\omega_c = \omega_z + \omega_y + \omega_x = \begin{bmatrix} \dot{\phi} - \sin(\psi)\dot{\varphi} \\ \cos(\phi)\dot{\psi} + \cos(\psi)\sin(\phi)\dot{\varphi} \\ -\sin(\phi)\dot{\psi} + \cos(\psi)\cos(\phi)\dot{\varphi} \end{bmatrix} \tag{4.12}$$

Rewriting the absolute angular velocity in terms of the generalized coordinates yields:

$$\omega_c = J\dot{q} = J \begin{bmatrix} \dot{\phi} \\ \dot{\psi} \\ \dot{\varphi} \end{bmatrix} \tag{4.13}$$

where J is the Jacobian from the rotation axes to the body frame:

$$J = \begin{bmatrix} 1 & 0 & -\sin(\psi) \\ 0 & \cos(\phi) & \cos(\psi)\sin(\phi) \\ 0 & -\sin(\phi) & \cos(\psi)\cos(\phi) \end{bmatrix} \tag{4.14}$$

Subsituting ω_c into the rotational part of the kinetic energy in (4.5) yields:

$$\frac{1}{2}\omega_c^T I_c \omega_c = [\dot{\phi}\ \ \dot{\psi}\ \ \dot{\varphi}]J^T I_c J \begin{bmatrix} \dot{\phi} \\ \dot{\psi} \\ \dot{\varphi} \end{bmatrix} = [\dot{\phi}\ \ \dot{\psi}\ \ \dot{\varphi}]\ \ M_r \begin{bmatrix} \dot{\phi} \\ \dot{\psi} \\ \dot{\varphi} \end{bmatrix} \tag{4.15}$$

where

$$M_r = \begin{bmatrix} I_x & 0 & -\sin(\psi)I_x \\ 0 & \cos(\phi)^2 I_y + \sin(\phi)^2 I_z & \cos(\psi)\sin(\phi)\cos(\phi)(I_y - I_z) \\ -\sin(\psi)I_x & \cos(\psi)\sin(\phi)\cos(\phi)(I_y - I_z) & \sin(\psi)^2 I_x + \cos(\psi)^2(\sin(\phi)^2 I_y + \cos(\phi)^2 I_z) \end{bmatrix} \tag{4.16}$$

The matrix M_r is the rotational part of the equivalent inertia matrix, M, which consists of two nonzero rectangular blocks:

$$M = \begin{bmatrix} \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix} & 0 \\ 0 & \begin{bmatrix} J_q \end{bmatrix} \end{bmatrix} \tag{4.17}$$

We are ready to derive the Corolis terms.

4.2 Coriolis Terms

Using (4.4), we obtain:

$$C_1 = \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \tag{4.18}$$

$$C_2 = \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \tag{4.19}$$

$$C_3 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.20}$$

$$C_4 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \cos(\phi)\sin(\phi)(I_y - I_z) & \cos(\psi)(\cos(2\phi)(I_z - I_y) - I_x) \\ 0 & 0 & \sin(\phi)\cos(\phi)\cos(\psi)^2(I_z - I_y) \end{bmatrix} \tag{4.21}$$

$$C_5 = \begin{bmatrix} 0 & 2\sin(\phi)\cos(\phi)(I_z - I_y) & \cos(\psi)(\cos(2\phi)(I_y - I_z) + I_x) \\ 0 & 0 & 0 \\ 0 & 0 & \sin(\psi)\cos(\psi)(-I_x + \sin(\phi)^2 I_y + (\cos(\phi))^2 I_z) \end{bmatrix} \tag{4.22}$$

$$C_6 = \begin{bmatrix} 0 & \cos\psi\cos(2\phi)(I_y - I_z) & 2\sin\phi\cos\phi\cos^2\psi(I_y - I_z) \\ -\cos\psi I_x & \sin(\psi)\sin\phi\cos\phi(I_z - I_y) & 2\sin\psi\cos\psi(I_x - \sin^2\phi I_y - \cos^2\phi I_z) \\ 0 & 0 & 0 \end{bmatrix} \tag{4.23}$$

4.3 Potential Energy

The potential energy of the vehicle is due to gravity only (the forces generated by the suspension springs will be treated separetly):

$$E_p = mgz \tag{4.24}$$

4.4 The Equations of Motion

Using (4.2), we can now write the right hand side of the equations of motion:

$$Q_1 = m\ddot{x} \tag{4.25}$$

$$Q_2 = m\ddot{y} \tag{4.26}$$

$$\mathbf{Q3} = m\ddot{z} - mg \tag{4.27}$$

$$\begin{aligned}\mathbf{Q4} = \; & I_x\ddot{\phi} - \sin(\psi)I_x\ddot{\varphi} \\ & + \cos(\phi)\sin(\phi)(I_y - I_z)\dot{\psi}^2 + \cos(\psi)(\cos(2\phi)(I_z - I_y) - I_x)\dot{\varphi}\dot{\psi} \\ & + \sin(\phi)\cos(\phi)\cos(\psi)^2(I_z - I_y)\dot{\varphi}^2\end{aligned} \tag{4.28}$$

$$\begin{aligned}Q_5 = \; & (\cos(\phi)^2 I_y + \sin(\phi)^2 I_z)\ddot{\psi} + \cos(\psi)\sin(\phi)\cos(\phi)(I_y - I_z)\ddot{\varphi} \\ & + \mathbf{2}\;\sin(\phi)\cos(\phi)(I_z - I_y)\dot{\phi}\dot{\psi} + \cos(\psi)(\cos(2\phi)(I_y - I_z) + I_x)\dot{\varphi}\dot{\phi} \\ & + \sin(\psi)\cos(\psi)(-I_x + \sin(\phi)^2 I_y + \cos(\phi)^2 I_z)\dot{\varphi}^2\end{aligned} \tag{4.29}$$

$$\begin{aligned}Q_6 = \; & -\sin(\psi)I_x\ddot{\phi} + \cos(\psi)\sin(\phi)\cos(\phi)(I_y - I_z)\ddot{\psi} \\ & + (\sin(\psi)^2 I_x + \cos(\psi)^2(\sin(\phi)^2 I_y + \cos(\phi)^2 I_z))\ddot{\varphi} \\ & + \cos(\psi)(-I_x + \cos(2\phi)(I_y - I_z))\dot{\psi}\dot{\phi} \\ & + \sin(\psi)\sin(\phi)\cos(\phi)(I_z - I_y)\dot{\psi}^2 + 2\sin(\phi)\cos(\phi)\cos(\psi)^2(I_y - I_z)\dot{\varphi}\dot{\phi} \\ & + 2\sin(\psi)\cos(\psi)(I_x - \sin(\phi)^2 I_y - \cos(\phi)^2 I_z)\dot{\varphi}\dot{\psi}\end{aligned} \tag{4.30}$$

4.5 Generalized Forces

We now derive the left hand side of (4.2) in terms of the external forces acting on the body. These forces consist of the longitudinal (tractive) and lateral forces developed by the rear tire, the lateral forces developed by the front tire, and the spring forces. The tire forces are assumed to be parallel to the horizonal plane of the inertial frame; the spring forces are assumed to be parallel to the z axis of the inertial frame.

The total exteral forces in the A (ground) frame are:

$$F_A = \begin{bmatrix} F_{tr} + F_{tl} - \sin(\beta)(F_{fr} + F_{fl}) \\ F_{rr} + F_{rl} + \cos(\beta)(F_{fr} + F_{fl}) \\ F_{zfr} + F_{zfl} + F_{zrr} + F_{zrl} \end{bmatrix} \tag{4.31}$$

The generalized forces corresponding to the translational motion are the projections of the total external forces on the directions of the generalized coordiantes. The generalized forces are thus obtained by projecting F_A into frame O:

$$Q_1 = \cos(\varphi)(F_{tr} + F_{tl} - \sin(\beta)(F_{fr} + F_{fl})) - \sin(\varphi)(F_{rr} + F_{rl} + \cos(\beta)(F_{fr} + F_{fl})) \tag{4.32}$$

$$Q\,2 = \sin(\varphi)(F_{tr} + F_{tl} - \sin(\beta)(F_{fr} + F_{fl})) + \cos(\varphi)(F_{rr}\,\mathrm{t}\,F_{rl} + \cos(\beta)(F_{fr} + F_{fl})) \tag{4.33}$$

$$Q3 = F_{zfr}\ \mathrm{t}\ F_{zfl}\ \mathrm{t}\ F_{zrr}\ \mathrm{t}\ F_{zrl} \tag{4.34}$$

The generalized forces corresponding to the rotational motion are derived by projecting the total external moment exerted on the body to the individual

axes of rotation corresponding to the generalized coordiantes.

Total moment due to tire forces in the inertial frame is:

$$M_O = \begin{vmatrix} (F_{zfl} + F_{zrl})l - (F_{zfr} + F_{zrr})l \\ (F_{zfr} + F_{zfl})d_f - (F_{zrr} + F_{zrl})d_r \\ -(F_{rl} + F_{rr})d_r + \cos(\beta)(F_{fr} + F_{fl})d_f + l\sin(\beta)(F_{fl} - F_{fr}) \end{vmatrix} \tag{4.35}$$

Projecting M_O to the individual axes of rotation yields:

$$\begin{aligned} Q_4 &= \cos(\psi)\cos(\varphi)l(F_{zfl} + F_{zrl} - F_{zfr} - F_{zrr}) + \cos(\psi)\sin(\varphi)(d_f(F_{zfr} + F_{zfl}) \\ &- d_r(F_{zrr} + F_{zrl})) + \sin(\psi)d_r(F_{rl} + F_{rr}) - \sin(\psi)\cos(\varepsilon)d_f(F_{fr} - F_{fl}) \\ &+ \sin(\psi)\sin(\beta)l(F_{fr} + F_{fl}) \end{aligned} \tag{4.36}$$

$$\begin{aligned} Q_5 &= \sin(\varphi)l(F_{zfr} + F_{zrr} - F_{zfl} - F_{zrl}) \\ &+ \cos(\varphi)(d_f(F_{zfr} + F_{zfl}) - d_r(F_{zrr} + F_{zrl})) \end{aligned} \tag{4.37}$$

(4.38)

$$Q_6 = -(F_{rl} + F_{rr})d_r + \cos(\beta)(F_{fr} + F_{fl})d_f + l\sin(\beta)(F_{fl} - F_{fr}) \tag{4.39}$$

Note that the numbering of the generlized forces corresponds to the order of the genealized coordiantes.

4.6 Tire Forces

$$F_{zfr} = k(z + h - \sin\psi d_f - \frac{\cos\psi\sin\phi l}{2} - \cos\psi\cos\phi h) + \frac{W}{4} \tag{4.40}$$

$$F_{zfl} = k(z + h - \sin\psi d_f + \frac{\cos\psi\sin\phi l}{2} - \cos\psi\cos\phi h) + \frac{W}{4} \tag{4.41}$$

$$F_{zrr} = k(z + h - \sin\psi d_r - \frac{\cos\psi \sin\phi l}{2} - \cos\psi \cos\phi h) + \frac{W}{4} \quad (4.42)$$

$$F_{zrl} = k(z + h - \sin\psi d_r + \frac{\cos\psi \sin\phi l}{2} - \cos\psi \cos\phi h) + \frac{W}{4} \quad (4.43)$$

The side forces are determined from the tire slip angles, assumed to be equal for the right and left tires

$$F_f r = -C_\alpha \alpha_f \quad (4.44)$$

$$F_f l = -C_\alpha \alpha_f \quad (4.45)$$

$$F_r r = -C_\alpha \alpha_r \quad (4.46)$$

$$F_r l = -C_\alpha \alpha_r \quad (4.47)$$

where

$$\alpha_r = tan^{-1}[\frac{-d_r\dot{\varphi} + \dot{y}\cos\varphi - \dot{x}\sin\varphi}{\dot{x}\cos\varphi + \dot{y}\sin\varphi}] \quad (4.48)$$

$$\alpha_f = tan^{-1}[\frac{d_f\dot{\varphi} + \dot{y}\cos\varphi - \dot{x}\sin\varphi}{\dot{x}\cos\varphi + \dot{y}\sin\varphi}] - \beta \quad (4.49)$$

This completes the derivation of all the elements in (4.2). For dynamic simulations, it remains to invert (4.2) to

$$\ddot{q} = M(q)^{-1}(Q - c(q,\dot{q}) + \frac{\partial E_p}{\partial q}) \quad (4.50)$$

Chapter 5

Summary

In this project, we have developed methods for generating emergency maneuvers for automated vehicles, considering a planar nonlinear vehicle model with three degrees-of-freedom, and two control inputs: the front steering angle and the rear tractive force. We considered two main approaches: 1) generating emergency maneuvers by computing the time optimal velocity profile along a specified path, so as to minimize reaction time, while ensuring that the maneuver is dynamically feasible, and hence safe, and 2) generating the shortest lane change maneuvers for obstacle avoidance.

Although considering only a simple planar vehicle model, this work provides invaluable insights into the dynamic behavior of ground vehicles, which might not be apparent when using separate lateral and longitudinal models.

This work is summarized in (Shiller and Sundar 1995, Shiller and Sundar 1996), and is implemented in a computer program on a Silicon Graphics workstation with interactive graphics.

5.1 Time-Optimal Maneuvers

To assist in evaluating the dynamic feasibility of a given maneuver, we establish the upper bounds on vehicle performance by solving the time optimal control problem along the specified path with free boundary speeds. This yields the velocity limit curve in the phase plane, $s - \dot{s}$, which serves as an upper bound on *all* feasible velocity profiles. Emergency maneuvers can thus be generated from any initial conditions by selecting a velocity profile that does not cross, but slides along, the velocity limit curve.

Dynamic analysis of the vehicle's performance shows significant deviations between the steady-state and transient motions. It is shown that the steering angle reaches an upper limit, derived from one of the force equations, during transient motions at relatively low speeds, whereas this limit is generally not active during steady-state motions. This suggests that steady-state information is inapplicable for preview control schemes at extreme conditions. The transient steering angle is shown to be discontinuous, following discontinuities in path curvature, and hence dynamically infeasible, considering the nonzero dynamics of the steering actuator. The transient steering is also shown to oscillate due to the oscillatory nature of the body slip angle, which depends on the vehicle speed.

The use of the velocity limit curve for generating emergency maneuvers is demonstrated for a lane change at a constant speed. Independently computing the highest constant speed maneuver that does not violate the control constraints resulted in a trajectory that is tangent to the lowest point of the velocity limit curve. Obviously, this maneuver could have been selected

without any computation simply by searching over the velocity limit curve. Other potential maneuvers, starting at higher speeds, can be generated by cruising at a constant speed, or accelerating, until hitting the velocity limit curve, then sliding along the velocity limit until completing the lane change.

5.2 Obstacle Avoidance

Optimal emergency maneuvers for obstacle avoidance are computed by minimizing the longitudinal distance of the lane transition. This produces the sharpest feasible maneuvers for given initial speeds, representing the upper bound on the vehicle's ability to avoid an obstacle at given initial speeds. The distance to the closest avoidable obstacle, depicted by the *clearing distance* curve, is shown to be linear with respect to the initial speed for the optimal maneuvers, compared to the hyperbolic curve for the distance to a full stop. This clearly demonstrates the advantage of the optimal lane transition over a full stop at high speeds.

The *clearing distance* curve was shown as a valuable design tool for planning safe avoidance maneuvers. It allows to maximize the time from the detection of the obstacle to the time when a lane change maneuver must be executed. It was shown that the maximum time spent in the current lane is achieved by decelerating at the maximum deceleration as soon as the obstacle is detected, until reaching the *clearing distance*. Maximizing the time in the current lane might be necessary for communicating with the neighboring vehicles and for ensuring that a space is open in the neighboring lane for the maneuvering vehicle. The *clearing distance* curve can also be used as a

bench-mark for evaluating other avoidance maneuvers, and for specifying the minimum range of obstacle detection sensors.

On-line computation of emergency maneuvers was also presented. We proposed two methods: the first is based on a simple point mass model, and the second is based on extrapolating maneuvers from an optimal maneuver, computed at some nominal initial speed. Both approaches are computationally very efficient, and both are shown to closely approximate the paths of the optimal maneuvers. The point mass model might require an adjustment of the velocity profile. These maneuvers can be used as reference inputs to on-line feedback controllers.

5.3 6D Dynamic Simulations

We attempted to validate the maneuvers generated by both methods in dynamic simulations of the 6D (3D with rotations) rigid body model and thus evaluate the significance of the unmodeled load shift. Unfortunately, these simulations did not agree well with our nominal trajectories, although the dynamic simulations using the planar model were quite close. This can be attributed either to the unmodeled dynamics, or to modeling (more likely programming) errors. Further investigation into this issue was out of the initial scope for this project, and hence was not pursued.

References

Allen. R.W., H.J., Szostak, T.J., Rosenthal, D.H., Klyde, and K.J.. Owens. 1991. "Characteristics influencing ground vehicle lateral/directional dynamic stability," SAE Paper No. 910234.

Bryson. A., Inverse Dynamic Optimization, MANE Seminar, UCLA, April 1995.

Bryson. A. E. and Ho, Y.C., 1969, *Applied Optimal Control*, Blaisdell Publishing Company.

Chee. W., and M., Tomizuka. 1994. "Lane Change Maneuver of Automobiles for the Intelligent Vehicle and Highway System (IVHS)," American Control Conference, pp. 3586-3587.

Dugoff. H., P.S. Fancher, L. Segel. 1970. "An Analysis of Tire Properties and Their Influence on Vehicle Dynamics Perforamce," SAE Transactions 700377.

Frere, P., 1992. *Sports Car and Competition Driving*, Robert Bentley Publishing, Cambridge, MA.

Godbole, D. and J. Lygeros, 1994, "Longitudinal Control of the Lead Car of a Platoon", *IEEE Transactions on Vehicular Technology*, Volume 43, Number 4, Pages 1125-1135.

Hessburg, T., and M., Tomizuka. 1991. "A Fuzzy Rule-Based Controller for Automotive Vehicle Guidance," PATH Research Report, UCB-ITS-PRR-91-18, August.

Hessburg, T., H., Peng, and M., Tomizuka. 1991. "An Experimental Study on Lateral Control of a Vehicle," American Control Conference, pp. 3084-3089.

Lunger, P., "The Influence of the Structure of Automobile Models and Tire Characteristics on the Theoretical Results of Stead-State and Transient Vehicle Performance," The Dynamics of Vehicle, Proceedings 5th VSD-2nd IUTAM Symposium.

Maalej, A.Y., Guenther, D.A., and J.R., Ellis. 1989. "Experimental Development of Tire Force and Moment Models," *International Journal of Vehicle Design*, Vol. 10, No. 1, pp. 34-50.

Narendran, V.K., and J.K., Hedrick. 1993. "Transition Maneuvers in Intelligent Vehicle Highway Systems," Proc. of Conference on Decision and Control, San Antonio, TX, December, pp. 1880-1884.

Peng, H., and M., Tomizuka. 1990. "Lateral Control of Front-Wheel-Steering Rubber-Tire Vehicles," PATH Report, UCB-ITS-PRR-90-5.

Peng, H., and M., Tomizuka. 1991. "Preview Control for Vehicle Lateral Guidance in Highway Automation", American Control Conference, pp. 3090-3095.

Sakai, H., 1981. "Theoretical and Experimental Studies on the Dynamic Properties of Tires," *International Journal of Vehicle Design*, Vol. 2, No. 1.

Seywald, H., "Trajectory Optimization Based on Differential Inclusion," *AIAA Journal of Guidance, Control and Dynamics*, Vol. 17, No. 3. 1994.

Sheikholeslam, S., and C.A., Desoer. 1992. "Combined longitudinal and lateral control of a platoon of vehicles," ACC, pp. 1763-1767.

Shiller, Z., and Dubowsky, S. "Time Optimal Path Planning for Robotic Manipulators with Obstacles, Actuator, Gripper and Payload, Constraints," *Int'l J. Robotics Research*, December 1989, pp 3-18.

Shiller, Z., and H.H., Lu. 1992. "Computation of Path Constrained Time Optimal Motions along Specified Paths," *ASME J. of Dynamic Systems, Measurement and Control*, Vol. 114, No. 3, pp. 34-40, March.

Shiller, Z., Sundar, S., "Emergency Maneuvers of AHS Vehicles," 1995 SAE Future Transportation Technology Conference, Costa Mesa, CA, August 1995, SAE Paper 951893, SP-1106.

Shiller, Z., Sundar, S., "Emergency Lane-Change Maneuvers," 1996 Proc. of IFAC, San Francisco, July 1996.

Smith, D.E., and J.M., Starkey. 1994. "Effects of model complexity on the performance of automated vehicle steering controller: controller development and evaluation," ***Journal of Vehicle System Dynamics, 33,*** pp. 627-645.

Wong, J.Y., ***Theory of Ground Vehicles,*** John Wiley & Sons, New York, 1978.

Optimal Emergency Maneuvers of Automated Vehicles

M.O.U. 125

Software

Zvi Shiller and Satish Sundar

Department of Mechanical, Aerospace and Nuclear Engineering

University of California Los Angeles

Los Angeles, CA 90095

The following is a hardcopy of the software developed during this project. These files make the "vehicle" program, which optimizes the motions of a ground vehicle (a bicycle model) along specified paths and between given boundary conditions.

The inputs are the control points of a B spline describing the path (this is used as the initial guess for the path optimization), and the vehicle parameters. The output is generated by a menu driven interface, which allows at the lowest level to compute inverse dynamics along a specified path and specified speeds (compute the control inputs for the specified trajectory). These speeds may be either constant, or specified by a B spline. The next step is to generate the velocity limit curve, which maximizes the feasible speeds along the given path. The next and last step is to optimize the path between given boundary conditions.

The program was developed on a Silicon Graphics 4D-70 (system 3.0). It might need some small modifications to run on newer SGI machines.

```
# Makefile for compilation
# PATH project

CFLAGS = -Zg -0
CLIBS  =
30BJS = vehicle.0 win.o bspline.o algeb.o drw_terrain.o gclpt_w.o \
dynamics.0 runge.o runge 2.0 runge_3.0 compute-alpha.0 compute-steering.0 \
simulate.0 runge 4.0 bspiine_vel.o opt_trajectory.o btopar.o get_time.o\
patsh.o simulate12.0 runge_sim.o simulate_3.o runge_sim_3.o bspline_w.o \
limitc.o get-opt-time-a1onggath.o BSPLINE.o BSPLINE_vel.o BSPLINE_w.o \
dugoff.o runge_dugoff.o

vehicle: $(3OBJS)
        f77 $(CFLAGS) -o vehicle $(3OBJS) $(CLIBS)

vehicle.0                   : vehicle.c
win.0                        : win.c
bspline.o                    : bspline.f
algeb.o                      : algeb.f
drw_terrain.o                : drw_terrain.c
gclpt_w.o                    : gclpt_w.c
dynamics.o                   : dynamics.c
runge.o                        runge.c
runge_2.o                    : runge_2.c
runge_3.o                    : runge_3.c
compute-alpha.0              : compute-a1pha.c
compute-steering.0           : compute-steering.c
simulate.0                   : simulate.c
runge_4.o                    : runge_4.c
bspline vel.o                  bspline vel.f
opt-trajectory.0             : opt-trajectory-c
btopar.o                     : btopar.c
get-time.0                   : get_time.c
patsh.o                      : patsh.f
simulate_ 2.0  .             : simulate_2.c
runge sim.o                  : runge_sim.c
simulate_3.0                   simulate-3.c
runge_sim_3.0                : runge_sim_3.c
bspline_w.o                  : bspline_w.f
limitc.o                     : limitc.c
get-opt-time-along-path.0    : get-opt-time-a1onggath.c
BSPLINE.o                    : BSPLINE.f
BSPLINE_vel.o                : BSPLINE vel.f
BSPLINE_w.o                  : BSPLINE-w.f
dugoff.o                     : dugoff.c
runge_dugoff.o               : runge_dugoff.c
```

main

```
/*
 *program: program to test the single link manipulator
 *history:
 */
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "runge.h"
#include "limitc.h"
#include "objects.h"
#define sqr(x) ((x)*(x))
#define abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define PI              3.141592654

struct p {int winposi[4][10];/*screen coords of the windows*/'
       float wincoord[6][10];/* limits on the local coordinate
                                        systems of the windows*/
       int wingid[10]; /*window ids*/
       int nwin;  /*total number of windows opened*/
                        } winpar ;
struct q {int centid;} id;/*stores the id of the centre window*/
struct r {int ppos[4];} blnk;/*screen coordinates of the currently
                                    blank window*/
struct u {float angx,angy,dx,dy; int px,py;} gr;

struct w {float l1,l2,l3,xc,yc,zc;} wdim;

struct y (float pos;} motion;/*number of the point at which the robot
                              is being drawn*/

struct s3 {int pos;} curr; /*number of point at which the robot is being
                                    displayed by pressing the ESC key*/

/*rotation about the z axes*/
float ang_z;

/*introduced for the acceleration lines*/
/**
struct dyn_par { float mas[2],lc[2],ll[2],inrt[2],q[2],amp[2];} dimensions_;
**/
float s[500],x[500][2],xs[500][2],xss[500][2];/*time optimization declns*/
float vel_s[1000],vel_x[1000][2],vel_xs[1000][2],vel_xss[1000][2];
                  /*for the velocity profile alogn the path*/
float accel_s[1000]; /*the acceleration sdd(s) along the path*/
float velpro_s[1000], velpro_sd[1000], velpro_sdd[1000];
struct t {int nconp; float conp[20][2];} bz_;
struct tw {int nconp; float conp[20][2];} bzw_;
struct tv {int nconv; float conv[20][2];} bzvel_;
int PATH_NO;
int VEL_PATH_NO;
float time;

/*for the bspline representing the path slope*/
float sangle[500],xangle[500][2],xangles[500][2],xangless[500][2];
/*initial value of theta*/
float init_point[2];

/*actual value of theta to check the accuracy of the path integration*/
float theta_actual[500];

/* car dimensions */
struct vehicle (float mass, Ic, d, alpha, alpha-f, alpha-r, Ft, Beta, fnr, fnf,
width, wheel-length, wheel-width; } car;
```

```
struct paths {float theta[500],theta_s[500],theta_ss[500];} path;
struct paths2 {float alpha[500],alpha_s[500],alpha_ss[500];} pdyn;
struct forces {float beta[1000], drive[1000], dis[1000]; int np;} force;

/*index of the current car position*/
int curr_pos_of-car;

/*constant velocity of car*/
float constant-velocity-of-car:

/*store flag for runge_2*/
int store-flag-for-runge-2;

/*to store the time along a given path*/
float time_along_path[1000];

/*flag to select path type*/
int path-type; /*0 for theta-s as bspline, 1 for x-s as bspline*/

int vel_type; /*0 for sdd-s as bspline, 1 for sd-s as bspline*/

/*acceleration or velocity profile when the switch is made in vel. type*/
float vel_profile_at_switch[400][2];
/*theta profile when the switch is made in path type*/
float pos_profile_at_switch[400][2];
int   pos_profile_no;

/*upper and lower limits on the steering angle*/
float Beta_max, Beta min;

/*desired initial and final thetas*/
float slope_0_desired,  slope_f_desired;

/*current integer increment*/
int   current_int_increment;


/*to simulate using different models*/
extern int model-flag;   /*0 for saturation model
                           1 and 2 for dugoff model    */

main()
{
FILE *fin;
FILE *fout;

long  xminvp,xmaxvp,yminvp,ymaxvp,
  id,wkcel,winyz,wintor,limitc,
  optcel,torqcel,win1,win2,win3,win4,sphw,sphw2,timcel,blank_id,xw1,xw2 ;
long tfrear, alrear;
float xminwin,xmaxwin,yminwin,ymaxwin,zminwin,zmaxwin;

short val;  /*value of qeued device*/
char c;   /*exit letter*/
Matrix m; /*matrix for transformation*/
long gwin,cent_win,curr_id,menu ; /*window declarations*/
int i,cent_pos[4],temp_pos[4],flag,sign; /*window declarations*/
int id_menu;/*number of pop up menu item*/
int cell-no:  /*window id number*/
long ier; /*error flag*/
int j,k; /*indices*/
int limc_flag; /*flag to plot limit curve*/


/*integration routines */
```

```
float get_quantity_at_s();

/*to test subroutines*/
float test-angle, test_C, test-force;
float f_fn();



/*************************************************************************/
/*    read the path type:                                                */
/*               0 --> represent theta(s) using Bspline                  */
/*               1 --> represent x(s)     using Bspline                  */
/*************************************************************************/
fin=fopen("path_type.dat","r");
fscanf(fin, "%d",&path_type) ;
fclose(fin);

/*************************************************************************/
/*    read the velocity type:                                            */
/*               0 --> represent sdd(s) using Bspline                    */
/*               1 --> represent sd(s)   using Bspline                   */
/*************************************************************************/
fin=fopen("vel_type.dat","r");
fscanf(fin, "%d",&vel_type);
fclose(fin);


/*************************************************************************/
/* read the control points representing theta along the path             */
/*************************************************************************/
fin=fopen("bj_theta.inp","r");
fscanf(fin,"%d",&bz_.nconp);
for(i=1;i<=bz_.nconp;++i)fscanf(fin,"%f %f",&bz_.conp[i-1][0],&bz_.conp[i-1][1]);
fclose(fin);

/*************************************************************************/
/* read the control points representing the path in the workspace        */
/*************************************************************************/
fin=fopen("bw_space.inp","r");
fscanf(fin,"%d",&bzw_.nconp);
for(i=1;i<=bzw_.nconp;++i)fscanf(fin,"%f %f",&bzw_.conp[i-1][0],&bzw_.conp[i-1][1])
fclose(fin);

/*************************************************************************/
/* read the initial point                                                */
/*************************************************************************/
fin=fopen("init_point.inp","r");
fscanf(fin,"%f %f",&init_point[0],&init_point[1]);
fclose(fin);

/*************************************************************************/
/* read the constant velocity at which vehicle travels                   */
/*************************************************************************/
fin=fopen("constant_velocity_of_car.inp","r");
fscanf(fin, "%f",&constant_velocity_of_car);
fclose(fin);

/*************************************************************************/
/*              read the dynamic parameters of the car                   */
/*              mass = mass of car                                       */
/*              Ic   = moment of inertia about CG                        */
/*              d    = half length of the body                           */
/*              width= width of body and wheel                           */
/*              wheel-length= length of the wheel                        */
/*              wheel_width= length of the wheel                         */
/*************************************************************************/
```

```
fin=fopen("car.dimensions","r");
fscanf(fin, "%f %f %f %f %f %f",&car.mass, &car.Ic, &car.d, &car.width, &car.wheel_l
fclose(fin);

/*check car dimensions*/
printf("car.mass = %f\n", car.mass);
printf("car.Ic = %f\n", car.Ic);
printf("car.d = %f\n", car.d);
printf("car.width = %f\n", car.width);
printf("car.wheel_length = %f\n", car.wheel_length);
printf("car.wheel_width = %f\n", car.wheel_width);

/*************************************************************************/
/*   read the control points representing the velocity profile along     */
/*   the path                                                            */
/*************************************************************************/

fin = fopen("vel_profile.inp","r");
fscanf(fin, "%d",&bzvel_.nconv);
for(i=1;i<=bzvel_.nconv;++i)fscanf(fin,"%f %f",&bzvel_.conv[i-1][0],&bzvel_.conv[i-
fclose(fin);

/*************************************************************************/
/*   read the desired initial and final values of theta for the path     */
/*   as a fraction of PI (for convenience)                               */
/*************************************************************************/

fin = fopen("desired_slopes.inp","r");
fscanf(fin, "%f %f",&slope_0_desired,&slope_f_desired);
fclose(fin);
slope_0_desired = slope_0_desired*PI;
slope_f_desired = slope_f_desired*PI;

/*************************************************************************/
/*   define the limits on the steering angle                             */
/*************************t************************************************/
Beta max = PI/3.0;
Betamin = -PI/3.0;

/*define the curr position of the car*/
curr_pos_of_car = 1;
car.Beta = 0.0; /*3*PI/2;*/
car.alpha = pdyn.alpha[0]; /*PI/6.0;*/

/*initially no central window*/
cent_win=0;

/*initialize*/
c = 's';

/************************
lower textport window
*********************/
textport(5,1000,5,200);
foreground();
winpar.nwin = 1;

/*****************************
oordinates of the central window
*****************************/
blnk.ppos[0] = 10;
blnk.ppos[1] = 855;
blnk.ppos[2] = 175;
blnk.ppos[3] = 1020;

/************************
```

```
 define windows
 ************************/
/*--- #1:workcell and robot----*/
xminvp = 865;
xmaxvp = 1065;
yminvp = 820;
ymaxvp = 1020;
xminwin = -1.0; /*-10.0;*/ /*-2.000;*/
xmaxwin = 135;  /*31.0;*/ /*10.0;*/ /*20.0;*/ /*120.00;*/ /*60.0;*/ /*15.0;*/ /*50.0(
yminwin = -15.5;  /*-10.0;*/ /*-2.0;*/ /*-7.5;*/ /*-2.000;*/
ymaxwin = 15.5;  /*10.0;*/ /*20.0;*/ /*120.00;*/ /*60.0;*/ /*7.5;*/ /*2.00;*/
zminwin = -5.000;
zmaxwin = 5.000;
defwin("single link",&win1,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmaxwin,&yminw:
doublebuffero;
overlay(Z);
gconfig() ;

/*--- #2:--- */

xminvp = 1075;
xmaxvp = 1275;
yminvp = 820;
ymaxvp = 1020;
xminwin = -0.20;
xmaxwin = 135.0;  /*40.0;*/ /*120.0;*/ /*15.0;*/ /*50.0;*/ /*old 4.2*/
yminwin = -0.1*PI; /*-1.2*PI;*/ /*-0.2;*/
ymaxwin  = 0.1*PI; /*1.2*PI;*/
zminwin = -5.0:
zmaxwin = 5.0;
defwin("velocity - time", &winyz,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmaxwin,&
doublebuffero;
overlay(Z);
gconfig() ;

/*--- #3:--- */
xminvp = 865;
xmaxvp = 1065;
yminvp = 610;
ymaxvp = 810;
xminwin = 0;
xmaxwin = 4;
yminwin = -4;
ymaxwin = 4;
zminwin = -1.0;
zmaxwin = 1.0;
defwin("acceleration - time",&winyz,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmaxw:
doublebuffero;
overlay(2);
gconfig() ;

/*---- X4: test potentials---- */
xminvp = 1075;
xmaxvp = 1275;
yminvp = 610;
ymaxvp = 810;
xminwin = -2.0;
xmaxwin = 135.0; /*40.0;*/ /*120.0;*/
yminwin = -5.500;
ymaxwin = 75.0; /*45.0;*/ /*35.000;*/
zminwin = -5.0;
zmaxwin = 5.0;
defwin("actuator - distance", &limitc,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmax
doublebuffero;
overlay(Z);
gconfig() ;
```

```
/*---- X5: th7 vs th8 ----*/
xminvp = 865;
xmaxvp = 1065;
yminvp = 400;
ymaxvp = 600;
xminwin = -4.0;
xmaxwin = 9.0;
yminwin = -4.0;
ymaxwin = 4.0;
zminwin = -5.0;
zmaxwin = 5.0;
defwin("",&optcel,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmaxwin,&yminwin,&ymaxw.
doublebuffero;
overlay(2);
gconfig() ;

/*---- #6 ----*/
xminvp = 1075;
xmaxvp = 1275:
yminvp = 400;
ymaxvp = 600;
xminwin = -0.1;
xmaxwin = 135.0; /*40.0;*/ /*800.0;*/ /***40.0;***/ /*120.0;*/ /*50.0;*/ /*50.0*/
yminwin = -0.3; /*-2.0;*/ /****-0.2;****/ /*-5.0; */
ymaxwin = 0.3; /*2.0;*/ /***0.2;***/ /*5.0; */
zminwin = -5.0;
zmaxwin = 5.0;
defwin("",&timcel,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmaxwin,&yminwin,&ymaxw.
doublebuffero;
overlay(Z);
gconfig() ;

/*---- #7 ----*/
xminvp = 865;
xmaxvp = 1065;
yminvp = 190;
ymaxvp = 390;
xminwin = -2.0;
xmaxwin = 135.0; /*40.0;*/ /*800.0;*/ /***40.0;***/ /*120.0;*/
yminwin = -PI/2.0 -0.1 ; /*-5.0;*/
ymaxwin = PI/2.0 + 0.1 ; /*5.0;*/
zminwin = -5.0;
zmaxwin = 5.0;
defwin("limitc",&win3,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmaxwin,&yminwin,&yı
doublebuffero;
overlay(2);
gconfig() ;

/*-- #8: optimal actuator torques/joint accelerations-- */
xminvp = 1075;
xmaxvp = 1275;
yminvp = 190;
ymaxvp = 390;
xminwin = -2.0;
xmaxwin = 135.0; /*40.0;*/ /*800.0;*/ /***40.0;***/
yminwin = -6100.0;
ymaxwin = 6100.0;
zminwin = -5.0;
zmaxwin = 5.0;
defwin("limitc",&win4,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmaxwin,&yminwin,&yı
doublebuffero;
overlay(Z);
gconfig() ;

/*---- #9 ----*/
```

```
xminvp = 865;
xmaxvp = 1065;
yminvp = 10;
ymaxvp = 180;
xminwin = -2.0;
xmaxwin = 135.0;  /*40.0;*/ /*800.0;*/ /***40.0;***/ /*120.0;*/
yminwin = -6100 ; /f-5.0;*/
ymaxwin = 6100 ; /*5.0;*/
zminwin = -5.0;
zmaxwin = 5.0;
defwin("limitc",&tfrear,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmaxwin,&yminwin,
doublebuffer();
overlay(2);

/*-- #10-- */
xminvp = 1075;
xmaxvp = 1275;
yminvp = 10;
ymaxvp = 180;
xminwin = -2.0;
xmaxwin = 135.0;  /*40.0;*/ /*800.0;*/ /***40.0;***/
yminwin = -6100.0;
ymaxwin = 6100.0;
zminwin = -5.0;
zmaxwin = 5.0;
defwin("limitc",&alrear,&xminvp,&xmaxvp,&yminvp,&ymaxvp,&xminwin,&xmaxwin,&yminwin,
doublebuffer();
overlay(2);
gconfig() ;


curr.pos =1; /*draw the robot at the initial position*/
motion.pos=  -3.35; /*to draw the motion dials for the initial point*/

/*define menus*/
menu =defpup("menu %t|OPTARM1|OPTARM2|Save Path Data|Test Runge-Kutta|Integrate alp)

/*initial cursor positions*/
gr.dx=blnk.ppos[0]+25;
gr.dy=blnk.ppos[0]+25;
gr.angx=0.0;
gr.angy=0.0;

/************************************
define workcell dimensions and centre
*****t*****************************/
wdim.l1=4.00;
wdim.l2=4.000;
wdim.l3=6.000;
wdim.xc=0.0;
wdim.yc=0.0;
wdim.zc=0.0;

/*generate the path profile*/
generate_path_profile(path_type);

/*generate velocity profile*/
generate_velocity_profile();

/*store the path in two files for matlba plotting*/
fout = fopen("pathx_out.mat","w");
for (i=1;i<=PATH_NO-1;++i) fprintf(fout,"%f\n",x[i-1][0]);
fclose(fout);

fout = fopen("pathy_out.mat","w");
for (i=1;i<=PATH_NO-l;++i) fprintf(fout,"%f\n",x[i-1][1]);

   /*near-index= find-nearest(gr.px,gr.py); */
```

```
fclose(fout);


/* draw the vehicle and path*/
drw_theta(2);

/* draw the vehicle and path*/
drw_path_w(1);

/*plot the steering angles and driving forces in window 7*/
current-int-increment = 1;
plot_steer_and_drive_forces(current_int_increment,7);

/* queue the mouse buttons and ESC key */
qdevice(LEFTMOUSE);
qdevice(RIGHTMOUSE);
qdevice(MIDDLEMOUSE);
qdevice(ESCKEY);

/*initialize  flag*/
store_flag_for_runge_2 = 0;

while(c=='s')
{
 switch(qread(&val))

  case LEFTMOUSE:
  gr.px=getvaluator(MOUSEX);
  gr.py=getvaluator(MOUSEY);
  getwin(&gr.px,&gr.py,&gwin);

  if ((gwin==5) II (gwin==7))
  {
  while(getbutton(LEFTMOUSE))

  /*get the cursor screen coodinates*/
  gr.px=getvaluator(MOUSEX);
  gr.py=getvaluator(MOUSEY);
  cell-no = gwin;
  /*rotbox(&cell_no); */
  }
  }

  if (gwin==2)/*change control pts in trnasform space*/
  {
   /*near-index= find-nearest(gr.px,gr.py); */
   gr.px=getvaluator(MOUSEX);
   gr.py=getvaluator(MOUSEY);
   while(getbutton(LEFTMOUSE))
   {
     gr.px=getvaluator(MOUSEX);
     gr.py=getvaluator(MOUSEY);
     gclpt_w(gwin);
     bspline2_(&PATH_NO,sangle,xangle,xangles,xangless,&ier) ;
     generate_path_profile(path_type);
     generate_velocity_profile();
     drw_path_w(1);
     drw_theta(2);
   }
  }

  if (gwin==4)/*change control pts for velocity profile*/

  '/*near_index= find_nearest(gr.px,gr.py); */
   gr.px=getvaluator(MOUSEX);
   gr.py=getvaluator(MOUSEY):
```

```
    while(getbutton(LEFTMOUSE))
    {
      gr.px=getvaluator(MOUSEX);
      gr.py=getvaluator(MOUSEY);
      gclpt_w(gwin);
      generate_velocity_profile();
    }
   }

   if (gwin==1)/*change control pts in trnasform space*/
   {
    /*near-index=  find-nearest(gr.px,gr.py);  */
    gr.px=getvaluator(MOUSEX);
    gr.py=getvaluator(MOUSEY);
    while(getbutton(LEFTMOUSE))
    {
      gr.px=getvaluator(MOUSEX);
      gr.py=getvaluator(MOUSEY);
      gclpt_w(gwin);
      generate_path_profile(path_type);
      generate_velocity_profile();
      drw_path_w(1);
      drw_theta(2);
    }
   }

   break;

   case ESCKEY:
   while(getbutton(ESCKEY))
   {
     if(curr_pos_of_car == PATH-NO) currgos-of-car = 1;
     else                    curr_pos_of_car = currgos-of-car + 1;
     drw_path_w(1);
   }
   break:

/************************************************
 if the middle mouse button is clicked
 ************************************************/
   case MIDDLEMOUSE:
   gr.px = getvaluator(MOUSEX);
   gr.py = getvaluator(MOUSEY);
   getwin(&gr.px,&gr.py,&gwin);


   flag =0;


   if(gwin>0)
   {
     winset(winpar.wingid[gwin-1]);
     curr_id = gwin;

     if(cent_win==0)
     {
        cent_win = gwin;/*gwin is central window*/
        for(i=1;i<=4;++i)
        {
          cent_pos[i-1]= winpar.winposi[i-1][gwin-1];/*save coordinates of current c
          winpar.winposi[i-1][gwin-1]=blnk.ppos[i-1];

         drawmode(OVERDRAW);
         color(BLACK);
         clear();
         drawmode(NORMALDRAW);
```

```
  swapbuffers();

  winposition(blnk.ppos[0],blnk.ppos[1],blnk.ppos[2],blnk.ppos[3]);
  reshapeviewport();
  color(BLACK);
  clear();
  swapbufferso;
  flag =1;
  gwin =0;
}
 if(flag==1) break;


 if(cent_win>0)
  {

      if(cent_win==gwin)
      {
         cent_win=0;

          drawmode(OVERDRAW);
          color(BLACK);
          clear();
          drawmode(NORMALDRAW);
          swapbufferso;

          winposition(cent_pos[0],cent_pos[1],cent_pos[2],cent-pos[3]);
          reshapeviewport();
          color(BLACK);
          clear();
          swapbufferso;

         for(i=1;i<=4;++i)
         {
         winpar.winposi[i-1][gwin-1]=cent_pos[i-1];
          cent_pos[i-1]=blnk.ppos[i-1];
          }
          flag = 1;
          gwin =0;
      }

         if(flag ==1) break;

         if(cent_win!=gwin)
         {

          winset(winpar.wingid[cent_win-1]);

          drawmode(OVERDRAW);
          color(BLACK);
          clear();
          drawmode(NORMALDRAW);
          swapbufferso;

          winposition(cent_pos[0],cent_pos[1],cent_pos[2],cent_pos[3]);
          reshapeviewport();
          color(BLACK);
          clear ();
          swapbufferso;

         for(i=1;i<=4;++i)
         {
         winpar.winposi[i-1][cent_win-1]=cent_pos[i-1];
          cent_pos[i-1]=blnk.ppos[i-1];
          }
          cent_win = gwin;
```

```
/* while(!getbutton (ESCKEY));*/

            winset(winpar.wingid[gwin-1]);
            drawmode(OVERDRAW);
            color(BLACK);
            clear();  .
            drawmode(NORMALDRAW);
            swapbuffers();

            winposition(cent_pos[0],cent_pos[1],cent_pos[2],cent_pos[3]);
            reshapeviewport();
            color(BLACK);
            clear();
            swapbuffers();

           for(i=1;i<=4;++i)
           {
           temp_pos [i-l]  = winpar.winposi[i-1][gwin-1];
           winpar.winposi[i-1][gwin-1]=cent_pos[i-1];
            cent_pos[i-1]=temp_pos[i-1] ;
            }

            gwin = 0;

           }



    }
  }
   flag =0;
   gwin =0;

/************************************/
/* redraw in the new cell positions */
/************************************/
 generate_path_profile(path_type);
 drw_theta(2);
 drw_path_w(1);
 generate-velocitygrofile():
 plot_steer_and_drive_forces(current_int_increment,7);
 plot_simul_results(); /*simulation results*/
 callobj(simul_alpha);
 break;


 case  RIGHTMOUSE:
   id_menu = dopup(menu);

   if(id_menu==1){
                   generate_velocity_profile();
                   /*mak_path(x,PATH_NO);
                   gen_path_data(PATH_NO,x,s,xs,xss);
                   optpathsub-(&PATH-NO,s,x,xs,xss,&time);
                   */
                   /*
                   for(i=1;i<=PATH_NO;++i)printf("tt(%d)=%f\n",i,
                   trjct_.tt [i-1]);
                   */
                   /*
                   drw_single_link(1);
                   plotm_(trjct_.tt,trjct_.sd,PATH_NO,2);
                   plotm_(trjct_.tt,trjct_.aco,PATH_NO,3);
                   plot_2_(s,trjct_.tor,1,PATH_NO,4);
                   */
                   }
```

```
if(id_menu==2){}
if(id_menu==3){/*save the path data*/
                fin=fopen("init_point.out","w");
                fprintf(fin,"%f %f ",init_point[0],init_point[1]);
                fclose(fin);

                fin=fopen("bj_theta.out","w");
                fprintf(fin,"%d\n",bz_.nconp);
                for(i=1;i<=bz_.nconp;++i)fprintf(fin,"%f %f ",
                                bz_.conp[i-1][0],bz_.conp[i-1][1]);
                fclose(fin);

                fin = fopen("vel_profile.out","w");
                fprintf(fin, "%d\n",bzvel_.nconv);
                for(i=1;i<=bzvel_.nconv;++i)fprintf(fin,"%f %f ",
                bzvel_.conv[i-1][0],bzvel_.conv[i-1][1]);
                fclose(fin);

                fin=fopen("bw_space.out","w");
                fprintf(fin,"%d\n",bzw_.nconp);
                for(i=1;i<=bzw_.nconp;++i)fprintf(fin,"%f %f ",
                                bzw_.conp[i-1][0],bzw_.conp[i-1][1]);
                fclose(fin);

                fin=fopen("constant_velocity_of_car.out","w");
                fprintf(fin, "%f",constant_velocity_of_car);
                fclose(fin);
               }
if(id_menu==4){/*testing runge-kutta integration*/
                /*initialization*/
                fin = fopen("runge_initial.inp","r");
                 fscanf(fin,"%f %f %f %f %f %d",
                 &t0,&tfinal,&x0[0],&x0[1],&tol,&trace);
                fclose(fin);
                printf("trace = %d\n",trace);

                x0[0] = init_point[0];
                x0[1] = init_point[1];
                printf("x0 = %f,%f\n", x0[0], x0[1]);
                tfinal= s[PATH_NO-1];
                printf("tfinal = %f\n", tfinal);
                getchar();

                runge(t0,tfinal,x0,tol,trace,  tout,yout,&npoints,&ier);
                     }
if(id_menu==5){/*testing runge-kutta integration for alpha*/
                /*initialization*/
                current_int_increment = 1;
                fin = fopen("runge_alpha.inp","r");
                 fscanf(fin, "%f %f-%f %f %f %d",
                 &t02,&tfinal2,&x02[0],&x02[1],&tol2,&trace2);
                fclose(fin);
                printf("trace = %d\n",trace2);

                /*x02[0] = -0.2 ;*/ /*alpha(0)*/
                /*x02[1] = 0.0 */     ; /*alpha-s(0)*/
                /*x02[1] = -sin(x02[0])/car.d - path.theta_s[0]; */
                printf("x02 = %f,%f\n", x02[0], x02[1]);
                tfinal2= s[PATH_NO-1];
                printf("tfinal2 = %f\n", tfinal2);
                getchar();

                store_flag_for_runge_2 = 1;
                runge_2(t02,tfinal2,x02,tol2,trace2,  tout2, yout2,&npoints2
```

```
                    /*compute alpha for graphics*/
                    get_alpha_along_path();
                    /*compute the steering angle*/
                    find_path_steer_angles(current_int_increment);
                    plot_steer_and_drive_forces(current_int_increment,7);
                    runge_3(t02,tfinal2,x02,tol2,trace2,  tout2,yout2,&npoints2
                            }
if(id_menu==6) {
                    printf("testing f_fn\n");
                    scanf("%f",&test-angle);
                    scanf("%f",&test_C);
                    test_force = f_fn(test_angle,test_C);
                    printf("angle,c,force = %f, %f, %f\n", test-angle, test_C,t
                    printf("testing get_quantity\n");
                    scanf("%f", &test-angle);
                    printf("angle = %f\n", 180*get_quantity_at_s("the",test_ang
                    printf("angles = %f\n", get_quantity_at_s("ths",test_angle)
                    printf("angless = %f\n", get-quantity at_s("tss",test_angle
                    printf("sd = %f\n", get-quantity-at-sT"sdv",test-angle)  );
                    printf("sdd = %f\n", get_quantity_at_s("sdd",test_angle));
                                                            }
if(id_menu==7)simulate();
if(id_menu==8){
                  printf("type of model = ?\n");
                  scanf("%d",&model-flag);
                  simulate_2();
                  }
if(id_menu==9)simulate_3();
if(id_menu==10){
                   /*read initial conditions for runge-kutta integration*/
                    fin = fopen("runge_alpha.inp","r");
                     fscanf(fin, "%f %f %f %f %f %d",
                     &t02,&tfinal2,&x02[0],&x02[1],&tol2,&trace2);
                    fclose(fin);
                    printf("trace = %d\n",trace2);

                    printf("x02 = %f,%f\n", x02[0], x02[1]);
                    tfinal2= s[PATH_NO-1];
                    printf("tfinal2 = %f\n", tfina12);
                    getchar();

                    opt_trajectory("vel");
                  }
if(id_menu==11){
                   /*read initial conditions for runge-kutta integration*/
                    fin = fopen ("runge_alpha.inp","r");
                     fscanf(fin, "%f %f %f %f %f %d",
                     &t02,&tfinal2,&x02[0],&x02[1],&tol2,&trace2);
                    fclose(fin);
                    printf("trace = %d\n",trace2);

                    printf("x02 = %f,%f\n", x02[0], x02[1]);
                    tfinal2= s[PATH_NO-1];
                    printf("tfinal2 = %f\n", tfina12);
                    getchar();

                    /*set model type to use dugoff ellipse*/
                    model-flag = 2;
                    opt_trajectory("env");
                  }
if(id_menu==12){
                   /*read initial conditions for runge-kutta integration*/
                    fin = fopen("runge_alpha.inp","r");
                     fscanf(fin, "%f %f %f %f %f %d",
                     &t02,&tfinal2,&x02[0],&x02[1],&tol2,&trace2);
                    fclose(fin);
```

```
                printf("trace = %d\n",trace2);

                printf("x02 = %f,%f\n", x02[0], x02[1]);
                tfinal2= s[PATH_NO-1];
                printf("tfinal2 = %f\n", tfinal2);
                getchar();

                /*opt_trajectory("pth");*/
                /*opt_trajectory("pt2");*/
                opt_trajectory("pt3");
              }
if(id_menu==13){
              /*read initial conditions for runge-kutta integration*/
               fin = fopen("runge_alpha.inp","r");
                fscanf(fin,"%f %f %f %f %f %d",
                &t02,&tfinal2,&x02[0],&x02[1],&tol2,&trace2);
               fclose(fin);
               printf("trace = %d\n",trace2);

               printf("x02 = %f,%f\n", x02[0], x02[1]);
               tfinal2= s[PATH_NO-1];
               printf("tfinal2 = %f\n", tfinal2);
               getchar();

               opt_trajectory("trj");
              }
if(id_menu==14){
              /*read initial conditions for runge-kutta integration*/
               fin = fopen("runge_alpha.inp","r");
                fscanf(fin,"%f %f %f %f %f %d",
                &t02,&tfinal2,&x02[0],&x02[1],&tol2,&trace2);
               fclose(fin);
               printf("trace = %d\n",trace2);

               printf("x02 = %f,%f\n", x02[0], x02[1]);
               tfinal2= s[PATH_NO-1];
               printf("tfinal2 = %f\n", tfinal2);
               getchar();

               /*opt_trajectory("TRJ"); */
               opt_trajectory("TR2");
              }
if(id_menu==15){
               path-type = ( (path_type==1) ? 0 : 1 );
               /*make sure both representations have
                                  same initial point*/
               if (path-type==0)
               {
                  init_point[0] = bzw_.conp[0][0];
                  init_point[1] = bzw_.conp[0][1];
               }
               else
               {
                  bzw_.conp[0][0] = init_point[0];
                  bzw_.conp[0][1] = init_point[1];
               }
               pos_profile_no = PATH-NO;
               for(i=1;i<=pos_profile_no;++i)
                {pos profile at switch[i-1][0] = s[i-1];
                 pos_profile_at_switch[i-1][1] = path.theta[i-1];}
              }
if(id_menu==16){
               vel_type = ( (vel_type==1) ? 0 : 1 );
               /*make sure both representations have
                                  same initial velocity*/
               if (vel_type==0)
```

```
                        {
                           constant_velocity_of-car = bzvel_.conv[0] [l];
                           for(i=1;i<=VEL_PATH_NO;++i)/*old vel_type=1*/
                            {vel_profile_at_switch[i-1][0] = velpro_s[i-1];
                             vel_profile_at_switch[i-1][1] = velpro_sd[i-1];}
                        }
                        else
                        {
                           bzvel_.conv[0][1] = constant_velocity_of-car;
                           for(i=1;i<=VEL_PATH_NO;++i)/*old vel_type=0*/
                            {vel_profile_at_switch[i-1][0] = velpro_s[i-1];
                             vel_profile_at_switch[i-1][1] = velpro_sdd[i-1];}
                        }

                       }
        if(id_menu==17){
                       /*read initial conditions for runge_kutta integration*/
                        fin = fopen("runge_alpha.inp","r");
                         fscanf(fin, "%f %f %f %f %f %d",
                         &t02,&tfinal2,&x02[0],&x02[1],&tol2,&trace2);
                        fclose(fin);
                        printf("trace = %d\n",trace2);

                        printf("x02 = %f,%f\n", x02[0], x02[ 1]);
                        tfinal2= s[PATH_NO-1];
                        printf("tfinal2 = %f\n", tfina12);
                        getchar();

                        opt_trajectory("cvl");
                      }
        if(id_menu==18){
                        /*compute kinematic beta*/
                        /*compute alpha-r along the path*/
                        /*compute alpha-f along the path*/
                       }
        if(id_menu==19)c = 't';
        break ;

    }

    qreset();

    }

    /*****
    exit
    *****/
     gexit() ;

     /***********************************
     restore textport to centre of screen
     ***********************************/
     textport(20,1250,200,1000);
     }



/*************************************************************/
/*        find angle made with x-axis by a vector            */
/*************************************************************/
float anglex( x,  y ) /*returns angle between -pi and pi*/
float x, y;
{
float gamma;

if ( abs(x) < 0.000001 )
```

```
{
  if ( y >= 0.0 )
    gamma = PI/2;
  else
    gamma = 3*PI/2;
}

if (  x > 0.0 )
{
  gamma = (float)atan((double)(abs(y)/abs(x)));
  if ( y >= 0.0 )
     gamma = gamma;
  else
     gamma = 2*PI - gamma;
}

if ( x < 0.0 )
{
  gamma = (float)atan((double)(abs(y)/abs(x)));
  if ( y >= 0.0 )
     gamma = PI - gamma;
  else
     gamma = PI + gamma;
}

/*modification to make angle lie betwee -180 and 180*/
if ( (gamma>PI) && (gamma<=2*PI) ) gamma = gamma - 2*PI;

/**printf("anglex = %f \n", 180*gamma/PI); **/
return(gamma);
}

float anglex_2( x,  y ) /*returns angle between 0 and 2*pi*/
float x, y;
{
float gamma;

if ( abs(x) < 0.000001 )
{
  if ( y >= 0.0 )
    gamma = PI/2;
  else
    gamma = 3*PI/2;
}

if (  x > 0.0 )
{
  gamma = (float)atan((double) (abs(y)/abs(x)));
  if ( y >= 0.0 )
     gamma = gamma;
  else
     gamma = 2*PI - gamma;
}

if ( x < 0.0 )
{
  gamma = (float)atan((double)(abs(y)/abs(x)));
  if ( y >= 0.0 )
     gamma = PI - gamma;
  else
     gamma = PI + gamma;
}

/**printf("anglex = %f \n", 180*gamma/PI); **/
return(gamma);
}
```

```
real-char(s,len,x)
char s[];
int len;
float x;
{
int i,j,k,c,n,m,negative=0;

if(x<0.)
  {
  negative=1;
  x = -x;
  }

i-0;
n=m=(int) x;

do {
   s[i++]= n%10 + '0';
   } while ((n/=10) >0);

if (negative) s[i++]='-';

/*reverse the interger part */
for(j=0,k=i-1;j<k;j++,k--)
   {
   c=s[ j];
   s[j]=s[k];
   s[k]=c;
   }

if (len !=0) s[i++]='.';
x=x-m;

/*decimal part*/
for(j=0;j<len;j++)
  {
  x=x*10;
  n=(int)x;
  s[i++]='0'+n;
  x=x-n;
  }
s[i]='\0';
}
```

```
/*************************************************************************
LABORATORY FOR ROBOTICS AND AUTOMATION,MANE,UCLA,1990
-------------------------------------------------
program: defwin.c
function: program to open a 3-d box at the screen coordinates
(xminvp,xmaxvp,yminvp,ymaxvp) with limits on the coordinate axes specified by:
 (xminwin,xmaxwin) for the x axis
 (yminwin,ymaxwin) for the y axis
 (zminwin,zmaxwin) for the z axes
created: june 1990
history:
*************************************************************************/
#include <gl.h>
#include <device.h>
     struct p (int winposi[4][10];/*positions of the windows in screen
                                   coordinates*/
               float wincoord[6][10];/*limits on the box coordinate axes*/
               int wingid[10]; /*window ids*/
               int nwin;  /*number of windows */
                       } winpar ;

  defwin(wint,id,xminvp,xmaxvp,yminvp,ymaxvp,xminwin,xmaxwin,yminwin,ymaxwin,zminwi
     char wint[] ;
     long *xminvp,*xmaxvp,*yminvp,*ymaxvp,*id;
     float *xminwin,*xmaxwin,*yminwin,*ymaxwin,*zminwin,*zmaxwin;
  {

     /************************************
      store the window positions and local
      coordinates
      ************************************/
     winpar.winposi[0][winpar.nwin-1] = *xminvp ;
     winpar.winposi[1][winpar.nwin-1] = *xmaxvp ;
     winpar.winposi[2][winpar.nwin-1] = *yminvp ;
     winpar.winposi[3][winpar.nwin-1] = *ymaxvp ;
     winpar.wincoord[0][winpar.nwin-1] = *xminwin;
     winpar.wincoord[1][winpar.nwin-1] = *xmaxwin;
     winpar.wincoord[2][winpar.nwin-1] = *yminwin;
     winpar.wincoord[3][winpar.nwin-1] = *ymaxwin;
     winpar.wincoord[4][winpar.nwin-1] = *zminwin;
     winpar.wincoord[5][winpar.nwin-1] = *zmaxwin;

     /************************************
      no borders on the windows
      ************************************/
     noborder();

     /************************************
      window position in screen coords
      ************************************/
     prefposition(*xminvp,*xmaxvp,*yminvp,*ymaxvp);

     /***************
      window id
      ***************/
     *id = winopen(wint);
     winpar.wingid[winpar.nwin-1] = *id;
     wintitle(wint) ;

     /**************et*********
      open the 3-d box
      ************************/
     ortho(*xminwin,*xmaxwin,*yminwin,*ymaxwin,*zminwin,*zmaxwin);

     color(BLACK) ;
```

```
  clear() ;
  swapbuffers();

  color(BLACK) ;
  clear() ;
  swapbuffers();
  winpar.nwin = winpar.nwin+l;
  }

getwin(x,y,gwin)
int *x,*y; /*screen coordinates of the mouse button*/
int *gwin; /*window id number*/
{
  int i,flag;
  flag =0;

  /*********************************************
    find the window which encloses the point
    (x,y)
    *********************************************/
  for(i=1;i<=winpar.nwin;++i)
  {
    if(((*x>=winpar.winposi[0][i-1]) && (*x<=winpar.winposi[1][i-1]))&&
    ((*y>=winpar.winposi[2][i-1])&&(*y<=winpar.winposi[3][i-1])))
    {
       *gwin=i;
       flag =1;
    }
  }

   /*********************************
    if (x,y) is not in any window
     set window number = 0
     *********************************/
   if(flag==0) *gwin = 0;


 }
```

```
/*****************************************************************************
written by Satish
program to simulate the motions along a specified path of a bicycle
*****************************************************************************/

#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "runge_sim.h"

#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define  PI              3.141592654

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;

extern struct vehicle {float mass, Ic, d, alpha, alpha-f, alpha_r, Ft, Beta,
                       fnr,  fnf, width,  wheel-length, wheel-width; } car;


extern float s[500],x[500][2],xs[500][2],xss[500][2];
extern int PATH NO;
extern struct paths {float theta[500],theta_s[500],theta_ss[500];} path;
extern struct paths2 {float alpha[500],alpha_s[500],alpha_ss[500];} pdyn;
extern struct forces {float beta[1000], drive[1000], dis[1000]; int np;] force:

extern float constant-velocity-of-car;

extern float velpro_s[1000], velpro_sd[1000], velpro_sdd[1000];
extern int VEL_PATH_NO;
extern float init_point[2];

/*to store the time along path*/
extern float time_along_path[1000];

simulate-30
{
 long ier;
 float find_time_along_path();

 /*initialization*/
 to5     = 0.0;
 tfinal5 = s[PATH_NO-1];
 to15    = 0.1;
 trace5 = 1;
 x05[0]  = init_point[0];                                  /* xl = x */
 x05[1]  = cos(path.theta[0]);                             /* x2 =  dx/ds */
 x05[2]  = init_point[1];                                  /* yl = y */
 x05[3]  = sin(path.theta[0]);                             /* y2 = dy/ds */
 x05[4]  = 0.0;/*path.theta[0] + pdyn.alpha[0]; */         /* phi1 = phi */
 x05[5]  = 0.0;/*(path.theta_s[0] + pdyn.alpha_s[0]);*/    /*phi2 = dphi/ds*/
 x05[6]  = velpro_sd[0];                                   /* sd(0)       */
 x05[7]  =  0.0;                                           /* s(0)        */

 printf("x05 = %f,%f,%f,%f,%f,%f,%f,%f\n", x05[0], x05[1], x05[2], x05[3], x05[4], :
 printf("tfinal5 = %f\n", tfinal5);
 getchar();
```

```
 runge_sim_3(t05,tfinal5,x05,tol5,trace5,  tout5,yout5,&npoints5,&ier);

}


/*************************************************************************/
/*  for the integration of the state (x,dx/dt,y,dy/dt,phi,dphi/dt) with    */
/*   respect to s given the steering angle and the driving force          */
/*************************************************************************/
state_sim_4(dist,x,xprime)
float dist,x[8],xprime[8];
{
 int   i;

 float get_control_at_s(); /*returns the steering angle and driving force*/
 float get_quantity_at_s();
 float anglex();
 float f_alpha_r();
 float f_fn();
 float get_distance_at_time();

 /*float dist;*/

 float sd at dist, sdd_at_dist;
 float theta-at dist;
 float alpha-atdist;
 float control_i_dir, control_j_dir, control-phi-dir:

 float alpha-front, alpha-rear;
 float Crear, Cfront; /*force constant on rear wheel*/

 float steering-angle, driving-force:
 float psi-front;

 Crear  = 80000;
 Cfront = -80000:


 /*********************************************************************/
 /* using x[0] = x, x[1]= dx/ds, x[2] = y, x[3] = dy/ds,              */
 /*       x[4] = phi, x[5] = dphi/ds, x[6] =sd, x[7] = s              */
 /* compute the state dynamics at the given state                     */
 /*********************************************************************/

 sd_at_dist        = x[6];

 theta-at-dist     = get_quantity_at_s("the",dist);

 alpha-at-dist     = x[4] - theta_at_dist;


 /*get the steering angle and driving force*/
 steering-angle    = get_control_at_s("str",dist);
 driving-force     = get_control_at_s("drv",dist);

 /*determine  alpha-front*/
 psi-front  = anglex( cos(alpha_at-dist),
                car.d*x[5] - sin(alpha_at_dist) );

 alpha_front = steering-angle - psi-front;

 /*determine  alpha-rear*/
 alpha rear = f_alpha_r(dist, sd_at_dist, alpha-at-dist, 0.0, x[5]*sd_at_dist);

 /*find the controls in the i and j and phi directions*/
```

```
control_i_dir =
(
   ( driving-force - f_fn(alpha_front,Cfront)*sin(steering_angle) )*cos(x[4])
 - ( f_fn(alpha_rear,Crear)+f_fn(alpha_front,Cfront)*cos(steering_angle) )*sin(x[4
  )/car.mass  ;

control_j_dir =
(
   ( driving_force - f_fn(alpha_front,Cfront)*sin(steering_angle))*sin(x[4])
 + ( f_fn(alpha_rear,Crear)+f_fn(alpha_front,Cfront)*cos(steering_angle) ) *cos(x[4
 )/car.mass  ;

control_phi_dir  =
   car.d*(
      f_fn(alpha_front,Cfront)*cos(steering_angle) - f_fn(alpha_rear,Crear)
         )/car.Ic;

sdd_at_dist = (
                driving-force*cos(alpha-at_dist)
              - f_fn(alpha_front,Cfront)*sin(alpha_at_dist+steering_angle)
              - f_fn(alpha_rear,Crear)*sin(alpha_at_dist)
              )/car.mass;


/*compute the derivatives*/

xprime[0] = x[1];
                    /* dx/ds = x_s          */

xprime[l] = (control_i_dir-x[1]*sdd_at_dist)/sqr(sd_at_dist);
                    /* d2x/d2s = (control_x-x_s*sdd)/sqr(sd)  */

xprime[2] = x[3];
                    /* dy/ds = ys*/

xprime[3] = (control_j_dir-x[3]*sdd_at_dist)/sqr(sd_at_dist);
                    /* d2y/ds2 = (control-y-y-s*sdd)/sqr(sd)*/

xprime[4] = x[5];
                    /* dphi/ds = phi-s*/

xprime[5] = (control_phi_dir-x[5]*sdd_at_dist)/sqr(sd_at-dist) ;
                    /* d2phi/d2s = (control-phi-phi-sTsdd)/sqr(sd)*/

xprime[6] = sdd_at_dist/x[6];
                    /* d(sd)/ds = sdd/sd */

xprime[7] = 1.0;
                    /* ds/ds = 1*/

}

state_sim_3(dist,x,xprime)
float dist,x[8],xprime[8];
{
 int   i;

 float get-control_at_s(); /*returns the steering angle and driving force*/
 float get_quantity_at_s();
 float anglex();
 float f_alpha_r();
 float f_fn();
 float f_alpha_dd();

 float sdd at_dist;
 float theta_at_dist, thetas_at_dist, thetass_at_dist;
```

```
float control_i_dir, control_j_dir;

float alpha-front, alpha-rear;
float Crear, Cfront; /*force constant on rear wheel*/

float steering-angle, driving-force:
float psi-front;

Crear  = 80000;
Cfront = -80000;


/*********************************************************************/
/* using x[0] = x, x[1]= dx/ds, x[2] = y, x[3] = dy/ds,              */
/*       x[4] = alpha, x [5] = dalpha/ds, x[6] =sd, x[7] = s         */
/* compute the state dynamics at the given state                     */
/*********************************************************************/

theta_at_dist     = get_quantity_at_s("the",dist);
thetas_at_dist    = get_quantity_at_s("ths",dist);
thetass_at_dist   = get_quantity_at_s("tss",dist);



/*get the steering angle and driving force*/
steering-angle    = get_control_at_s("str",dist);
driving-force     =  get-control-at-s("drv",dist);

/*determine alpha-front*/
psi-front = anglex( cos(x[4]),
               car.d*(thetas_at_dist+x[5]) - sin(x[4]) );

alpha-front = steering-angle - psi-front;

/*determine alpha-rear*/
alpha-rear = f_alpha_r(dist, x[6], x[4], x[5]*x[6], thetas_at_dist*x[6]);

/*find the controls in the i and j and phi directions*/
control_i_dir =
t
   ( driving_force -
     f_fn(alpha_front,Cfront)*sin(steering_angle) )*cos(theta_at_dist+x[4])
 - ( f_fn(alpha_rear,Crear)+f_fn(alpha_front,Cfront)*cos(steering_angle) )*sin(thet
 )/car.mass  ;

control_j_dir =
t
   ( driving-force - f_fn(alpha_front,Cfront)*sin(steering_angle) )*sin(theta_at_d
 + ( f_fn(alpha_rear,Crear)+f_fn(alpha_front,Cfront)*cos(steering_angle) )*cos(thet
)/car.mass  ;


sdd_at_dist = (
                driving_force*cos(x[4])
              - f_fn(alpha_front,Cfront)*sin(x[4]+steering_angle)
              - f_fn(alpha_rear,Crear)*sin(x[4])
              )/car.mass;


/*compute the derivatives*/

xprime[0] = x[1];
                    /* dx/ds = x_s         */

xprime[1] = (control_i_dir-x[1]*sdd_at_dist)/sqr(x[6]);
xprime[1] = -sin(theta_at_dist)*thetas_at_dist;
```

```
                    /*  d2x/d2s  =  (control_x-x_s*sdd)/sqr(sd)  */

xprime[2] = x[3];
                    /* dy/ds = ys*/

xprime[3] = (control_j_dir-x[3]*sdd_at_dist)/sqr(x[6]);
xprime[3] = cos(theta_at_dist)*thetas_at_dist;
                    /* d2y/ds2  =  (control-y-y-s*sdd)/sqr(sd)*/

xprime[4] = x[5];
                    /* dalpha/ds = alpha-s*/

xprime[5] = f_alpha_dd(dist, x[6], sdd_at_dist, x[4], x[5]*x[6],
                       thetas_at_dist*x[6],
                       thetas_at_dist*sdd_at_dist + thetass_at_dist*sqr(x[6]),
                       Crear);

xprime[5] = ( xprime[5] - x[5]*sdd_at_dist )/sqr(x[6]);
                                                /* alpha-ss = . . .        */

xprime[6] = sdd_at_dist/x[6];
                    /* d(sd)/ds = sdd/sd */

xprime[7] = 1.0;
                    /* ds/ds = 1*/
```

```
/*****************************************************************************
written by Satish
program to simulate the motions along a specified path of a bicycle
*****************************************************************************/

#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "runge_sim.h"

Xdefine norm(x,y) sqrt(sqr(x)+sqr(y))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define  PI              3.141592654

extern struct p (int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;

extern struct vehicle {float mass, Ic, d, alpha, alpha-f, alpha r, Ft, Beta,
                       fnr, fnf, width, wheel-length, wheel_width; } car;


extern float s[500],x[500][2],xs[500][2],xss [500][2];
extern int PATH-NO;
extern struct paths {float theta[500],theta_s[500],theta_ss[500];} path;
extern struct paths2 {float alpha[500],alpha_s[500],alpha_ss[500];} pdyn;
extern struct forces {float beta[1000], drive[1000], dis[1000]; int np;} force;

extern float constant-velocity-of-car;

extern float velpro_s[1000], velpro_sd[1000], velpro_sdd[1000];
extern int VEL_PATH_NO;
extern float init_point[2];

/*to store the time along path*/
extern float time_along_path[1000];

/*to simulate using different models*/
int model-flag;    /*0 for saturation model
                     1 and 2 for dugoff model    */

/*dugoff model ellipse major and minor axes values for plotting?*/
extern float dugoff_ellipse_Fxmax,
             dugoff_ellipse_Fymax;
simulate-20
{
 FILE *fin;
 long ier;
 float find_time_along_path();

 int i, j;
 float anglex();
 float f_alpha_r();
 float sd dis;
 float drive-force;
 float get_control_at_s();
 float dugoff_force();
 float f_fn();
 float Crear:
```

```
/*initialization*/
to4     = 0.0;
/*
tfinal4 = find_time_along_path();
*/
tfina14 = s[PATH_NO-1];
to14    = 0.1;
trace4 = 1;
x04[0]  = init_point[0];                                    /* xl = x */
x04[1]  = cos(path.theta[0])*velpro_sd[0];                  /* x2 = dx/dt */
x04[2]  = init_point[1];                                    /* yl = y */
x04[3]  = sin(path.theta[0])*velpro_sd[0];                  /* y2 = dy/dt */
x04[4]  = path.theta[0] ;                          /* phi1 = phi */
x04[5]  = path.theta_s[0]*velpro_sd[0];                     /*phi2 = dphi/dt*/

printf("x04 = %f,%f,%f,%f,%f,%f\n", x04[0], x04[1], x04[2], x04[3], x04[4], x04[5]
printf("tfinal4 = %f\n", tfina14);
getchar();

/* read the type of model used*/
/*
fin=fopen("model_flag.dat","r");
 fscanf(fin, "%d",&model_flag);
fclose(fin);
*/

if(model_flag == 0)
  runge_sim(t04,tfinal4,x04,tol4,trace4,  tout4,yout4,&npoints4,&ier);
else
  runge_dugoff(t04,tfinal4,x04,tol4,trace4,  tout4,yout4,&npoints4,&ier);

/*--------transfer values to global variables for plotting ------------------*/
if(model_flag==1)
{
simul.np = npoints4;
for(i=1;i<=simul.np;++i)
  for(j=1;j<=STD2;++j) simul.x[i-1][j-1] = yout4[i-1][j-1]; /*STD2=6*/

for(i=1;i<=simul.np;++i) simul.dis[i-1] = tout4[i-1];

/*compute alpha*/
for (i=1;i<=simul.np;++i)
{
 simul.alpha[i-1] = -(yout4[i-1][4]-anglex(yout4[i-1][1],yout4[i-1][3]));

 /*compute  alpha-rear*/
 sd-dis = sqrt( sqr(yout4[i-1][1]) + sqr(yout4[i-1][3]) );
 simul.alpha_rear[i-1] =
    f_alpha_r(simul.dis[i-1], sd-dis, simul.alpha[i-1], 0.0, yout4[i-1][5]);

 /*compute linear model forces*/
 drive-force     = get_control_at_s("drv",simul.dis[i-1]);

 /*compute dugoff forces*/
 simul.dugoff_f_rear[i-1] = dugoff_force(drive_force,simul.alpha_rear[i-1]);

 /*store the major axes of the ellipse for plotting*/
 simul.dugoff_Fxmax[i-1] = dugoff_ellipse_Fxmax;
 simul.dugoff_Fymax[i-1] = dugoff_ellipse_Fymax;

 /*compute rear tire forces given by saturation model*/
 Crear = 80000.0;
 simul.sat_f_rear[i-1] = f_fn(simul.alpha_rear[i-1],Crear);
1
/*-------------------------------------------------------------------------*/
```

```
plot_simul_results();
}
}


/*************************************************************************/
/*   for the integration of the state (x,dx/dt,y,dy/dt,phi,dphi/dt) with  */
/*   respect to s given the steering angle and the driving force         */
/*************************************************************************/
state_sim_2(time,x,xprime)
float time,x[6],xprime[6];
{
 int   i;

 float get_control_at_s(); /*returns the steering angle and driving force*/
 float get_quantity_at_s();
 float anglex();
 float f_alpha_r();
 float f_fn();
 float get_distance_at_time();

 float dist;

 float sd_at_dist;
 float theta at dist;
 float alpha-at-dist;
 float control_i_dir, control_j_dir, control_phi_dir;

 float alpha-front, alpha-rear;
 float Crear,  Cfront; /*force constant on rear wheel*/

 float steering-angle, driving_force;
 float psi-front;

 Crear  = 80000;
 Cfront = -80000;


 /*find the correpsonding distance at the given time*/
 /*dist              = get-distance-at-time(time); */
 dist              = time;

 /*sd = sqrt( sqr(x2) + sqr(y2) )*/
 sd_at_dist        =  sqrt( sqr(x[1]) + sqr(x[3]) );

 theta-at-dist     = anglex(x[1],x[3]); /*get_quantity_at_s("the",dist);*/

 alpha_at_dist     = x[4] - theta-at-dist;

 /*********************************************************************/
 /* using x[0] = x, x[1]= dx/dt, x[2] = y, x[3] = dy/dt,              */
 /*       x[4] = phi,  x[5] = dphi/dt                                 */
 /* compute the state dynamics at the given state                     */
 /*********************************************************************/

 /*get the steering angle and driving force*/
 steering-angle    = get_control_at_s("str",dist);
 driving-force     = get_control_at_s("drv",dist);

 /*determine alpha-front*/
 psi-front = anglex( cos(alpha_at_dist),
                car.d*x[5]/sd_at_dist - sin(alpha_at_dist) );
                                        /*note: phi-s = x[5]/sd_at_dist*/
 alpha-front = steering-angle - psi-front;
```

```
 /*determine alpha-rear*/
 alpha-rear = f_alpha_r(dist, sd_at_dist, alpha-at-dist, 0.0, x[5]);

 /*find the controls in the i and j and phi directions*/
 control_i_dir =
 (
    ( driving-force - f_ fn(alpha_front,Cfront)*sin(steering_angle))*cos(x[4])
  - ( f_fn(alpha_rear,Crear)+f_fn(alpha_front,Cfront)*cos(steering_angle) )*sin(x[4
  )/car.mass   ;

 control_j_dir =
 t
    ( driving-force - f_fn(alpha_front,Cfront)*sin(steering_angle) )*sin(x[4])
  + ( f_fn(alpha_rear,Crear)+f_fn(alpha_front,Cfront)*cos(steering_angle) )*cos(x[4
 )/car.mass    ;

 controlphi-dir  =
    car.d*(
       f_fn(alpha_front,Cfront)*cos(steering_angle) - f_fn(alpha_rear,Crear)
          )/car.Ic;


 /*compute the derivative*/
 xprime[0] = x[1];                    /* dx1/dt = x2         */

 xprime[1] = control_i_dir;           /* dx2/dt = control-x */

 xprime[2] = x[3];                    /* dy1/dt = y2*/

 xprime[3] = control_j_dir ;          /* dy2/dt = control-y*/

 xprime[4] = x[5];                    /* dy1/dt = y2*/

 xprime[5] = control_phi_dir ;        /* dy2/dt = control-y*/

 for (i=1;i<=6;++i) xprime[i-1] = xprime[i-1]/sd_at_dist;


}

/*************************************************************************/
/*   for the integration of the state (x,dx/dt,y,dy/dt,phi,dphi/dt) with    */
/*   respect to s given the steering angle and the driving force           */
/*   using the dugoff model                                                */
/*************************************************************************/
state-dugoff(time,x,xprime)
float time,x[6],xprime[6];
{
 int   i:

 float get_control_at_s(); /*returns the steering angle and driving force*/
 float get_quantity_at_s();
 float anglex();
 float f_alpha_r();
 float f_fn();
 float get_distance_at_time();

 float dugoff_force();   /*computes rear tire force using dugoff model*/

 float dist;

 float sd at dist;
 float theta-at  dist;
 float alpha-at-dist;
 float control_i_dir, control_j_dir, control-phi-dir;
```

```
float alpha-front, alpha-rear;
float Crear, Cfront;  /*force constant on rear wheel*/

float steering-angle, driving-force;
float  psi-front;

Crear   = 80000;
Cfront = -80000:


/*find the correpsonding distance at the given time*/
/*dist              = get-distance-at-time(time); */
dist              = time;

/*sd = sqrt( sqr(x2) + sqr(y2) )*/
sd_at_dist        =  sqrt( sqr(x[1]) + sqr(x[3]) );

theta-at-dist     = anglex(x[1],x[3]); /*get_quantity_at_s("the",dist);*/

alpha-at-dist     = x[4] - theta-at-dist;

/*********************************************************************/
/* using x[0] = x, x[1]= dx/dt, x[2] = y, x[3] = dy/dt,              */
/*       x[4] = phi,  x[5] = dphi/dt                                 */
/* compute the state dynamics at the given state                     */
/*********************************************************************/

/*get the steering angle and driving force*/
steering-angle    = get_control_at_s("str",dist);
driving-force     = get_control_at_s("drv",dist);

/*determine alpha-front*/
psi-front = anglex( cos(alpha_at_dist),
              car.d*x[5]/sd_at_dist - sin(alpha_at_dist) );
                                          /*note: phi-s = x[5]/sd_at_dist*/
alpha-front = steering-angle - psi-front;

/*determine alpha-rear*/
alpha_rear = f_alpha_r(dist, sd_at_dist, alpha_at_dist, 0.0, x[5]);

/*find the controls in the i and j and phi directions*/
control_i_dir =
(
   ( driving-force - f_fn(alpha_front,Cfront)*sin(steering_angle) )*cos(x[4])
 - ( dugoff_force(driving_force,alpha_rear)+f_fn(alpha_front,Cfront)*cos(steering_
 )/car.mass  ;

control_j_dir =
(
   ( driving_force - f_fn(alpha_front,Cfront)*sin(steering_angle) )*sin(x[4])
 + ( dugoff_force(driving_force,alpha_rear)+f_fn(alpha_front,Cfront)*cos(steering_
)/car.mass  ;

control_phi_dir  =
   car.d*(
      f_fn(alpha_front,Cfront)*cos(steering_angle) - dugoff_force(driving_force,alp
         )/car.Ic;


/*compute the derivative*/
xprime[0] = x[1];                    /* dx1/dt = x2         */

xprime[1] = control_i_dir;           /* dx2/dt = control-x */

xprime[2] = x[3];                    /* dy1/dt = y2*/
```

```
 xprime[3] = control_j_dir ;          /* dy2/dt = control_y*/

 xprime[4] = x[5];                    /* dy1/dt = y2*/

 xprime[5] = control_phi_dir ;        /* dy2/dt = control_y*/

 for (i=1;i<=6;++i) xprime[i-1] = xprime[i-1]/sd_at_dist;


}



float find_time_along_path()
{
 float time-taken;

 int   i;

 time_along_path[0] = 0.0; /*initialze*/

 for (i=1;i<=VEL_PATH_NO-1; ++i)
 {
  if (velpro_sd[i-1]<0.01)  velpro_sd[i-1] = 0.01;

  time-along-path[i] = time_along_path[i-1] +
                       (velpro_s[i]-velpro_s[i-1])/velpro_sd[i-1] ;
 }

 return(time_along_path[VEL_PATH_NO-1]);
}

/*************************************************************************/
/*       returns the distance along the path at a given time             */
/*************************************************************************/
float get_distance_at_time(time)
float time;
{
int i, t-index;
float quantity;

int time-flag;

time-flag = 0;

/*find the indices of the points encompassing the given distance*/
for (i=1;i<=VEL_PATH_NO-1;++i)
{
  if( (time>=time_along_path[i-1]) && (time<time_along_path[i]) ) (t-index = i; tim
}

if (time-flag==l)
{
  return(
  velpro_s[t_index-1] +
  (time-time_along_path[t_index-1])*(velpro_s[t_index]-velpro_s[t_index-1])/(time_a
          );
}
else /*return quantities at the end of the velocity profile*/
{
  return(velpro_s[VEL_PATH_NO-1]);
}

}
```

```
/*****************************************************************************/
/*     plots the simulated results                                           */
/*****************************************************************************/
plot_simul_results()
{
 long  cell-id; /*cell id*/
 int   i;



/*--------------plot rear tire forces in cell no 9----------------------------*/
 cell-id = winpar.wingid[9-1];
 winset(cell_id);
 color (BLACK);
 clear();

 /*draw the axes*/
 color (RED);
 move(0.0,0.0,0.0);
 draw(150.0,0.0,0.0);
 move(0.0, -6500.0,0.0);
 draw(0.0,6500.0,0.0);

 linewidth(1);
 color(YELLOW);
 move(simul.dis[0],simul.dugoff_f_rear[0],0.0);
 for(i=2;i<=simul.np;++i) draw(simul.dis[i-1],simul.dugoff_f_rear[i-1],0.0);

 color(GREEN);
 move(simul.dis[0],simul.sat_f_rear[0],0.0);
 for(i=2;i<=simul.np;++i) draw(simul.dis[i-1],simul.sat_f_rear[i-1],0.0);

 swapbuffers();
/*-----------------------------------------------------------------------------*/

}
```

```
/*****************************************************************************
written by Satish
program to simulate the motions along a specified path of a bicycle
*****************************************************************************/

#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "runge_sim.h"

#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define PI              3.141592654

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;

extern struct vehicle {float mass, Ic, d, alpha, alpha-f, alpha r, Ft, Beta,
                         fnr, fnf, width, wheel-length, wheel_width; } car;


extern float s[500],x[500][2],xs[500][2],xss[500][2];
extern int PATH-NO;
extern struct paths {float theta[500],theta_s[500],theta_ss[500];} path;
extern struct paths2 {float alpha[500],alpha_s[500],alpha_ss[500];} pdyn;
extern struct forces {float beta[1000], drive[1000], dis[1000]; int np;} force;

extern float constant_velocity_of_car;

extern float velpro_s[1000], velpro_sd[1000], velpro_sdd[1000];
extern int VEL_PATH_NO;

simulate()
{
 FILE *fin;
 long ier;

 /*initialization*/
 fin = fopen("runge_simulate.inp","r");
  fscanf(fin, "%f %f %f %f %f %f %f %d",
  &t03,&tfinal3,&x03[0],&x03[1],&x03[2],&x03[3],&tol3,&trace3);
 fclose(fin);
 printf("trace = %d\n",trace3);

 x03[1] = velpro_sd[0]; /*constant-velocity-of-car;  */

 printf("x03  = %f,%f,%f,%f\n", x03[0], x03[1], x03[2], x03[3]);
 tfinal3= s[PATH_NO-1];
 printf("tfinal3 = %f\n", tfina13);
 getchar();

 runge_4(t03,tfinal3,x03,tol3,trace3,  tout3,yout3,&npoints3,&ier);

}


/*****************************************************************************/
/*   for the integration of the state (sd, alpha, alpha s) with       */
/*   respect to s given the steering angle and the driving force           */
/*****************************************************************************/
```

```
state_sim(dist,x,xprime)
float dist,x[4],xprime[4];
{
 float get_quantity_at_s();
 float get_control_at_s(); /*returns the steering angle and driving force*/
 float anglex();
 float f_alpha_r();
 float f_fn();
 float f_alpha_dd();

 float theta_at-dist, thetas_at-dist, thetass_at_dist;
 float thetad_at_dist, thetadd_at_dist;
 float sdd-at-dist;


 float alpha-front, alpha-rear;
 float Crear, Cfront; /*force constant on rear wheel*/

 float steering-angle, driving-force;
 float psi-front;

 Crear  = 80000;
 Cfront = -80000;


 /*find all required quantities at s = dist*/
 theta at_dist     = get_quantity_at_s("the",dist);
 thetas_at_dist    = get_quantity_at_s("ths",dist);
 thetass_at_dist   = get-quantity-at-s("tss",dist);

 /*********************************************************************/
 /* using s = x[0], sd = x[1], alpha = x[2], alpha-s = x[3],          */
 /* compute the state dynamics at the given state                     */
 /*********************************************************************/

 /*get the steering angle and driving force*/
 steering-angle    = get_control_at_s("str",dist);
 driving-force     = get_control_at_s("drv",dist);

 /*determine alpha-front*/
 psi-front = anglex( cos(x[2]),
                car.d*(thetas_at_dist+x[3]) - sin(x[2]) );
 alpha-front = steering-angle - psi-front;

 /*determine alpha-rear*/
 alpha_rear =  f_alpha_r(dist, x[1], x[2], x[3]*x[1], thetas_at_dist*x[1]) ;


 sdd-at-dist = (
               driving_force*cos(x[2]) -
               f_fn(alpha_rear,Crear)*sin(x[2]) -
               f_fn(alpha_front,Cfront)*sin( x[2]+steering_angle )
               )/car.mass  ;


 /*compute the derivative*/
 xprime[0] = 1.0;                                  /*  ds/ds =1          */

 xprime[1] = sdd_at_dist/x[1];                     /* d(sd)/ds =  sdd/sd  */

 xprime[2] = x[3];                                 /* d(alpha)/ds = alpha-s*/

 xprime[3] = f_alpha_dd(dist, x[1], sdd-at-dist,  x[2], x[3]*x[1],
                        thetas_at_dist*x[1],
                        thetas_at_dist*sdd_at_dist + thetass_at_dist*sqr(x[1]),
                        Crear);
```

```
 xprime[3] = ( xprime[3] - x[3]*sdd_at_dist )/sqr(x[1]);
                                                /* alpha-ss = . . .       */
}


/*************************************************************************/
/*   returns the controls at dist                                        */
/*************************************************************************/
float get-control-at-s(qty,dist)
char  qty[3];
float dist;
{
 int i,  s index:
 float  quantity;
 int    point-flag;

 point-flag = 0;
 /*find the indices of the points encompassing the given distance*/
 for (i=1;i<=force.np-1;++i)
 {
   if ( (dist>=force.dis[i-1]) && (dist<force.dis[i]) ) { s-index = i; point-flag =
 }

 if (point-flag == 1)
 {
   if  ( (qty[0]=='s') && (qty[1]=='t') && (qty[2]=='r') )
   {
     /*if qty = "str" return beta*/
     quantity = force.beta[s_index-1] +
     (dist-force.dis[s_index-1])*(force.beta[s_index]-force.beta[s_index-1])/(force
     return(quantity);
   }
   else  if  ( (qty[0]=='d') && (qty[1]=='r') && (qty[2]=='v') )
   {
     /*if qty = "drv" return theta-s*/
     quantity = force.drive[s_index-1] +
     (dist-force.dis[s_index-1])*(force.drive[s-index]-force.drive[s_index-1])/(for
     return(quantity);
   }
   else
   {
     printf("quantity unrecognizable!! ERROR !\n");
     getchar();
     return(-1000.0);
   }
 }
 else /*return quantities at the end of the path*/
 {
   if  ( (qty[0]=='s') && (qty[1]=='t') && (qty[2]=='r') )
   {
     quantity = force.beta[force.np-1];
     return(quantity);
   }
   else  if  ( (qty[0]=='d') && (qty[1]=='r') && (qty[2]=='v') )
   {
     quantity = force.drive[force.np-1];
     return(quantity);
   }
   else
   {
     printf("quantity unrecognizable!! ERROR !\n");
     getchar();
     return(-1000.0):
   }
```

```
/*this program uses the runge kutta method to integrate a differential equation
 */
 /* used for integration in the simulation*/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define abs(x) ( (x) >= 0.0 ? (x): -(x) )
#define sqr(x) ((x)*(x))
#define NP 1000 /* NP = no of points along the path*/
#define STD3 8     /*state dimension for the simulation*/

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;


runge_sim_3(t0,tfinal,y0,tol,trace,  ttout,yyout,npoints,ier)
float t0,tfinal,y0[STD3],tol;
int trace,*npoints,*ier;
float ttout[NP], yyout[NP][STD3];
{
FILE *fout;

float pow,quant;
int nargin;
float t;
float k1[STD3],k2[STD3],k3[STD3],k4[STD3],k[STD3], vec[STD3];
float hmax, hmin, h;
float y[STD3], tau, delta;
int icount,i;
long  cell-id; /*cell id*/

/*functions called from outside */
/*float norm(), infnorm(), power-(), min2();*/

*ier = 0;

/*check*/
if(trace){
   printf("in ode23\n");
   printf("t0 = %f, tfinal = %f\n",t0,tfinal);
   printf("y0 = %f,   %f, %f, % f ,  % f , %f, %f, %f\n",y0[0],y0[1],y0[2],y0[3],y0[4],y0
   printf("tol=%f\n",tol);
   printf("trace = %d\n",trace);
         }

/*initialization*/
t = to;
hmax = (tfinal-t)/5.0;
hmin = (tfinal-t)/20000.0;
h = (tfinal-t) /500;
if(trace) printf("at start, h= %f\n",h);
y[0] = y0[0]; y[1] = y0[1];
y[2] = y0[2]; y[3] = y0[3];
y[4] = y0[4]; y[5] = y0[5];
y[6] = y0[6]; y[7] = y0[7];
if(trace) printf("at start, y= %f,  %f, %f, %f, %f, % f ,  % f , %f\n",y[0],y[1],y[2],  y
icount = 1;
ttout[icount-1] = t;
```

```
yyout[icount-1][0] = y[0]; yyout[icount-1][1] = y[1];
yyout[icount-1][2] = y[2]; yyout[icount-1][3] = y[3];
yyout[icount-1][4] = y[4]; yyout[icount-1][5] = y[5];
yyout[icount-1][6] = y[6]; yyout[icount-1][7] = y[7];

/*main loop*/
while( t<tfinal  )
{
  if(t+h>tfinal){ h = tfinal -t;}

  /*slopes*/
  /*  kl  =  h*f(to,yo) */
  state_sim_3(t,y,  k1);
  for (i=1;i<=STD3;++i) k1[i-1] = h*k1[i-1];

  /*  k2  =  h*f(to+0.5*h,yo+1/2*k2) */
  for(i=1;i<=STD3;++i) vec[i-1] = y[i-1] + 0.5*k1[i-1];
  state_sim_3(t+0.5*h, vec,  k2);
  for (i=1;i<=STD3; ++i) k2[i-1] = h*k2[i-1];

  /* k3 = h*f(to+0.5*h, y0+1/2*k2)    */
  for(i=1;i<=STD3;++i) vec[i-1] = y[i-1] + 0.5*k2[i-1];
  state_sim_3(t+0.5*h, vec, k3);
  for (i=1;i<=STD3; ++i) k3[i-1] = h*k3[i-1];

  /* k4 = h*f(to+h, yo +k3)           */
  for(i=1;i<=STD3;++i) vec[i-1] = y[i-1] + k3[i-1];
  state_sim_3(t+h, vec, k4);
  for (i=1;i<=STD3; ++i) k4[i-1] = h*k4[i-1];

  for(i=1;i<=STD3;++i) k[i-1] = (k1[i-1]+2*k2[i-1]+2*k3[i-1]+k4[i-1])/6.0;

  /*update y */
  for(i=1;i<=STD3;++i) y[i-1] = y[i-1] + k[i-1];
  t = t+h;
  icount = icount +1;
  if(trace) printf("icount = %d\n", icount);
  ttout[icount-1] = t;
  for(i=1;i<=STD3;++i) yyout[icount-l][i-1] = y[i-1];
  if(trace) printf("y = %f,  %f, %f, %f, %f, %f, %f, %f,   t  = %f\n", y[0], y[1], y[2

}

if( t< tfinal ) {printf("singularity likely\n"); *ier =1;}
*npoints = icount;

/*plot the path in the first window*/
cell-id = winpar.wingid[1-1];
winset(cell_id);
linewidth(3);
frontbuffer(1);

color (WHITE);
move(yyout[0] [0], yyout[0][2], 0.0);
for (i=2;i<=*npoints;++i) draw(yyout[i-1][0], yyout[i-1][2], 0.0);

/*plot the velocity profile in the fourth window*/
cell-id = winpar.wingid[4-1];
winset(cell_id);
frontbuffer(1);

color (WHITE);
move(ttout[0], yyout[0][6], 0.0);
for (i=2;i<=*npoints;++i)  draw(ttout[i-11, yyout[i-1][6], 0.0);
linewidth(1);
```

```
/*plot alpha in the sixth window*/
cell-id = winpar.wingid[6-1];
winset(cell_id);
frontbuffer(1);

color(WHITE);
move(ttout[0], yyout[0][4], 0.0);
for (i=2;i<=*npoints;++i) draw(ttout[i-1], yyout[i-1][4], 0.0);

}
```

```
/*this program uses the runge kutta method to integrate a differential equation
 *
 */
 /* used for integration in the simulation*/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define abs(x) ( (x) >= 0.0 ? (x): -(x) )
#define sqr(x) ((x)*(x))
#define NP 1000 /* NP = no of points along the path*/
#define STD2 6     /*state dimension for the simulation*/

#include "objects.h"

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;


runge_sim(t0,tfinal,y0,tol,trace,  ttout,yyout,npoints,ier)
float t0,tfinal,y0[STD2],tol;
int trace,*npoints,*ier;
float ttout[NP], yyout[NP][STD2];
{
FILE *fout;
float anglex();

float  pow,quant;
int nargin;
float t;
float k1[STD2],k2[STD2],k3[STD2],k4[STD2],k[STD2], vec[STD2];
float hmax, hmin, h;
float y[STD2], tau, delta;
int icount,i;
long  cell-id; /*cell id*/

/*functions called from outside */
/*float norm(), infnorm(), power-(), min2();*/

*ier = 0;

/*check*/
if(trace) {
   printf("in ode23\n");
   printf("t0 = %f,  tfinal = %f\n",t0,tfinal);
   printf("y0 = %f,   %f, %f, %f, %f,  %f\n",y0[0],y0[1],y0[2],y0[3],y0[4],y0[5]);
   printf("tol=%f\n",tol);
   printf("trace = %d\n",trace);
         }

/*initialization*/
t = to;
hmax = (tfinal-t)/5.0;
hmin = (tfinal-t)/20000.0;
h = (tfinal-t)/500;
if(trace) printf("at start, h= %f\n",h);
y[0] = y0[0]; y[1] = y0[1];
y[2] = y0[2]; y[3] = y0[3];
y[4] = y0[4]; y[5] = y0[5];
if(trace) printf("at start, y= %f,  %f, %f, %f, %f, %f\n",y[0],y[1], y[2], y[3], y[4
icount = 1;
```

```
ttout[icount-1] = t;
yyout[icount-1] [01 = y[0]; yyout[icount-1][1] = y[1];
yyout[icount-1][2] = y[2]; yyout[icount-1][3] = y[3];
yyout[icount-1][4] = y[4]; yyout[icount-1][5] = y[5];

/*main loop*/
while( t<tfinal )
{
  if(t+h>tfinal){ h = tfinal -t; }

  /*slopes*/
  /*  kl  =  h*f(to,yo) */
  state_sim_2(t,y,  k1);
  for (i=1;i<=STD2;++i) k1[i-1] = h*k1[i-1];

  /*  k2  =  h*f(to+0.5*h,yo+1/2*k2) */
  for(i=1;i<=STD2;++i) vec[i-1] = y[i-1] + 0.5*k1[i-1];
  state_sim_2(t+0.5*h, vec,  k2);
  for (i=1;i<=STD2;++i) k2[i-1] = h*k2[i-1];

  /* k3 = h*f(to+0.5*h, y0+1/2*k2)    */
  for(i=1;i<=STD2;++i) vec[i-1] = y[i-1] + 0.5*k2[i-1];
  state_sim_2(t+0.5*h, vec, k3);
  for (i=1;i<=STD2; ++i) k3[i-1] = h*k3[i-1];

  /* k4 = h*f(to+h, yo +k3)           */
  for(i=1;i<=STD2;++i) vec[i-1] = y[i-1] + k3[i-1];
  state_sim_2(t+h, vec, k4);
  for (i=1;i<=STD2;++i) k4[i-1] = h*k4[i-1];

  for(i=1;i<=STD2;++i) k[i-1] = (k1[i-1]+2*k2[i-1]+2*k3[i-1]+k4[i-1])/6.0;

  /*update y */
  for(i=1;i<=STD2;++i) y[i-1] = y[i-1] + k[i-1];
  t = t+h;
  icount = icount +1;
  if(trace) printf("icount = %d\n", icount);
  ttout[icount-1] = t;
  for(i=1;i<=STD2;++i) yyout[icount-1] [i-l] = y [i-l];
  if(trace) printf("y = %f, %f, %f, %f, %f, %f,   t = %f\n", y[0], y[1], y[2], y[3],

}

if( t< tfinal ) {printf("singularity likely\n"); *ier =1;}
*npoints = icount;

/*plot the path in the first window*/
makeobj(simul_alpha);
cell-id = winpar.wingid[1-1];
winset(cell_id);
linewidth(3);
frontbuffer(1);

color(WHITE);
move(yyout[0][0], yyout[0][2], 0.0);
for (i=2;i<=*npoints;++i) draw(yyout[i-1][0], yyout[i-1][2], 0.0);
closeobj();

callobj (simul_alpha);

/*plot the velocity profile in the fourth window*/
cell-id = winpar.wingid[4-1];
winset(cell_id);
frontbuffer(1);

color(WHITE);
```

```
move(ttout[0], sqrt( sqr(yyout[0][1]) + sqr(yyout[0][3])), 0.0);
for (i=2;i<=*npoints;++i) draw(ttout[i-1],
                               sqrt( sqr(yyout[i-1][1]) + sqr(yyout[i-1][3])),
                               0.0);
linewidth(1);

/*plot alpha in the 6th window*/
cell-id = winpar.wingid[6-1];
winset(cell_id);
frontbuffer(1);

color(WHITE);
move(ttout[0], -(yyout[0][4]-anglex(yyout[0][1],yyout[0][3])), 0.0);
for (i=2;i<=*npoints;++i)
   draw(ttout[i-1], -(yyout[i-1][4]-anglex(yyout[i-1][1],yyout[i-1][3])), 0.0);

/*store s in data files for plotting*/
fout = fopen("s_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n", ttout[i-1]);
fclose(fout);

/*store sd in data files for plotting*/
fout = fopen("sd_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",
                                  sqrt ( sqr (yyout [i-l] [1]) + sqr (yyout [i-1][3]) ));
fclose(fout);

/*store alpha in data files for plotting*/
fout = fopen("alpha_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",
                           -(yyout[i-1][4]-anglex(yyout[i-1][1],yyout[i-1][3])));
fclose(fout);


/*store path in data files for plotting*/

fout = fopen("pathx_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",yyout[i-1][0]);
fclose(fout);

fout = fopen("pathy_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",yyout[i-1][2]);
fclose(fout);

}
```

```
/*this program uses the runge kutta method to integrate a differential equation
 */
 /* used for integration in the simulation using the dugoff model*/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define abs(x) ( (x) >= 0.0 ? (x): -(x) )
#define sqr(x) ((x)*(x))
#define NP 1000  /* NP = no of points along the path*/
#define STD2 6     /*state dimension for the simulation*/

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;


runge_dugoff(t0,tfinal,y0,tol,trace,  ttout,yyout,npoints,ier)
float t0,tfinal,y0[STD2],tol;
int trace,*npoints,*ier;
float ttout[NP], yyout[NP][STD2];
{
FILE *fout;
float anglex();

float pow,quant;
int nargin;
float t;
float k1[STD2],k2[STD2],k3[STD2],k4[STD2],k[STD2], vec[STD2];
float hmax, hmin, h;
float y[STD2], tau, delta;
int icount,i;
long  cell-id; /*cell id*/

/*functions called from outside */
/*float norm(), infnorm(), power-(), min2();*/

*ier = 0;

/*check*/
if(trace){
   printf("in ode23\n");
   printf("t0 = %f, tfinal = %f\n",t0,tfinal);
   printf("y0 = %f,   %f, %f, %f, %f, %f\n",y0[0],y0[1],y0[2],y0[3],y0[4],y0[5]);
   printf("tol=%f\n",tol);
   printf("trace = %d\n",trace);
         }

/*initialization*/
t = to;
hmax = (tfinal-t)/5.0;
hmin = (tfinal-t)/20000.0;
h = (tfinal-t)/500;
if(trace) printf("at start, h= %f\n",h);
y[0] = y0[0]; y[1] = y0[1];
y[2] = y0[2]; y[3] = y0[3];
y[4] = y0[4]; y[5] = y0[5];
if(trace) printf("at start, y= %f, %f, %f, %f, %f, %f\n",y[0],y[1], y[2], y[3], y[4
icount = 1;
ttout[icount-1] = t;
```

```
yyout[icount-l] [0] = y[0]; yyout[icount-1][1] = y[1];
yyout[icount-1][2] = y[2]; yyout[icount-1][3] = y[3];
yyout[icount-1][4] = y[4]; yyout[icount-1][5] = y[5];

/*main loop*/
while( t<tfinal  )
{
  if(t+h>tfinal){ h = tfinal -t; }

  /*slopes*/
  /*  kl = h*f(to,yo) */
  state_dugoff(t,y,  k1);
  for (i=1;i<=STD2;++i) k1[i-1] = h*k1[i-1];

  /*  k2  =  h*f(to+0.5*h,yo+1/2*k2) */
  for(i=1;i<=STD2;++i) vec[i-1] = y[i-1] + 0.5*k1[i-1];
  state_dugoff(t+0.5*h, vec,  k2);
  for (i=1;i<=STD2;++i) k2[i-1] = h*k2[i-1];

  /* k3 = h*f(to+0.5*h, y0+1/2*k2)   */
  for(i=1;i<=STD2;++i) vec[i-1] = y[i-1] + 0.5*k2[i-1];
  state_dugoff(t+0.5*h, vec, k3);
  for (i=1;i<=STD2;++i) k3[i-1] = h*k3[i-1];

  /* k4 = h*f(to+h, yo +k3)          */
  for(i=1;i<=STD2;++i) vec[i-1] = y[i-1] + k3[i-l];
  state_dugoff(t+h, vec, k4);
  for (i=1;i<=STD2;++i) k4[i-1] = h*k4[i-1];

  for(i=1;i<=STD2;++i) k[i-1] = (k1[i-1]+2*k2[i-1]+2*k3[i-1]+k4[i-1])/6.0;

  /*update y */
  for(i=1;i<=STD2;++i) y[i-1] = y[i-1] + k[i-1];
  t = t+h;
  icount = icount +1;
  if(trace) printf("icount = %d\n", icount);
  ttout[icount-l] = t;
  for(i=1;i<=STD2;++i) yyout[icount-1] [i-l] = y[i-1];
  if(trace) printf("y = %f, %f, %f, %f, %f, %f, t = %f\n", y[0], Y[1], Y[2], y[3],

}

if( t< tfinal ) {printf("singularity likely\n"); *ier =1;}
*npoints = icount;

/*plot the path in the first window*/
cell-id = winpar.wingid[1-1];
winset(cell_id);
linewidth(3);
frontbuffer(1);

color(WHITE);
move(yyout[0][0], yyout [0][2],0.0);
for (i=2;i<=*npoints;++i) draw(yyout[i-1][0], yyout[i-1][2], 0.0);

/*plot the velocity profile in the fourth window*/
cell-id = winpar.wingid[4-1];
winset(cell_id);
frontbuffer(1);

color(WHITE);
move(ttout[0], sqrt( sqr(yyout[0][1]) + sqr(yyout[0][3]) ), 0.0);
for (i=2;i<=*npoints;++i) draw(ttout[i-1],
                               sqrt( sqr(yyout[i-1] [1]) + sqr(yyout[i-1][3]) ),
                               0.0);
linewidth(1);
```

```
/*plot alpha in the 6th window*/
cell_id = winpar.wingid[6-1];
winset(cell_id);
frontbuffer(1);

color(WHITE);
move(ttout[0], -(yyout[0][4]-anglex(yyout[0][1] ,yyout [0][3])), 0.0);
for (i=2;i<=*npoints;++i)
   draw(ttout[i-1], -(yyout[i-1][4]-anglex(yyout[i-1][1],yyout[i-1][3])), 0.0);

/*store s in data files for plotting* /
fout = fopen("s_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n", ttout[i-1]);
fclose(fout);

/*store sd in data files for plotting*/
fout = fopen("sd_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",
                              sqrt( sqr(yyout[i-1][1]) + sqr(yyout[i-1][3])));
fclose(fout);

/*store alpha in data files for plotting*/
fout = fopen("alpha_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",
                            -(yyout[i-1][4]-anglex(yyout[i-1][1],yyout[i-1][3])));
fclose(fout);


/*store path in data files for plotting*/

fout = fopen("pathx_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",yyout[i-1][0]);
fclose(fout);

fout = fopen("pathy_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",yyout [i-l][2]);
fclose(fout);

}
```

```
/*this program uses the runge kutta method to integrate a differential equation
 */
 /* used for integration in the simulation*/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define abs(x) ( (x) >= 0.0 ? (x): -(x) )
#define sqr(x) ((x)*(x))
#define NP 1000 /* NP = no of points along the path*/
#define STD 4     /*state dimension for the simulation*/

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;


runge_4(t0,tfinal,y0,tol,trace,  ttout,yyout,npoints,ier)
float t0,tfinal,y0[STD],tol;
int trace,*npoints,*ier;
float ttout[NP], yyout[NPl [STD];
{
FILE *fout;

float pow,quant;
int nargin;
float t;
float k1[STD],k2[STD],k3[STD],k4[STD]1,k[STD], vec[STD];
float hmax, hmin, h;
float y[STD], tau, delta;
int icount,i;
long  cell-id; /*cell id*/

/*functions called from outside */
/*float norm(), infnorm(), power-(), min2();*/

*ier = 0;

/*check*/
if(trace){
   printf("in ode23\n");
   printf("t0 = %f, tfinal = %f\n",t0,tfinal);
   printf("y0 = %f,   %f, %f, %f\n",y0[0],y0[1],y0[2],y0[3]);
   printf("tol=%f\n",tol);
   printf("trace = %d\n",trace);
         }

/*initialization*/
t = to;
hmax = (tfinal-t)/5.0;
hmin = (tfinal-t)/20000.0;
h = (tfinal-t)/500;
if(trace) printf("at start, h= %f\n",h);
y[0] = y0[0]; y[1] = y0[1];
y[2] = y0[2]; y[3] = y0[3];
if(trace)printf("at start, y= %f, %f, %f, %f\n",y[0],y[1], y[2], Y[3]);
icount = 1;
ttout[icount-1] = t;
yyoutticount-11 [0] = y[0]; yyout[icount-1] [1] = y[1];
yyout[icount-1][2] = y[2]; yyout[icount-1][3] = y[3];

/*main loop*/
```

```
while( t<tfinal  )
{
  if(t+h>tfinal){ h = tfinal -t; }

  /*slopes*/
  /*  kl  =  h*f(to,yo) */
  state_sim(t,y,   k1);
  for (i=1;i<=STD;++i) k1[i-1] = h*k1[i-1];

  /*  k2  =  h*f(to+0.5*h,yo+1/2*k2) */
  for(i=l;i<=STD;++i)  vec[i-1] = y[i-1] + 0.5*k1[i-1];
  state_sim(t+0.5*h, vec,  k2);
  for (i=1;i<=STD;++i) k2[i-1] = h*k2[i-1];

  /* k3 = h*f(to+0.5*h, y0+1/2*k2)    */
  for(i=l;i<=STD;++i)  vec[i-1] = y[i-1] + 0.5*k2[i-1];
  state_sim(t+0.5*h, vec, k3);
  for (i=1;i<=STD;++i) k3[i-1] = h*k3[i-1];

  /* k4 = h*f(to+h, yo +k3)           */
  for(i=1;i<=STD;++i) vec[i-1] = y[i-1] + k3[i-1];
  state_sim(t+h,  vec, k4);
  for (i=1;i<=STD;++i) k4[i-1] = h*k4[i-1];

  for(i=1;i<=STD;++i) k[i-1] = (k1[i-1]+2*k2[i-1]+2*k3[i-1]+k4[i-1])/6.0;

  /*update y */
  for(i=1;i<=STD;++i) y[i-1] = y[i-1] + k[i-1];
  t = t+h;
  icount = icount +1;
  if(trace) printf("icount = %d\n", icount);
  ttout[icount-1] = t;
  for(i=1;i<=STD;++i) yyout[icount-1] [i-l] = y[i-1];
  if(trace) printf("y = %f, %f, %f, %f,   t = %f\n", y[0], y[1], y[2], y[3], t);

}

if( t< tfinal ) {printf("singularity likely\n"); *ier =1;}
*npoints = icount;

/*
printf("before drawing\n");
getchar();
*/
cell-id = winpar.wingid[6-1];
winset(cell_id);
linewidth(1);
frontbuffer(1);

/*plot the integrated function*/
color(WHITE);
move(ttout[0], yyout[0][2], 0.0);
for  (i=2;i<=*npoints;++i) draw(ttout[i-1], yyout[i-1][2], 0.0);

/*
color(YELLOW);
move(ttout[0],yyout[0][1], 0.0);
for (i=2;i<=*npoints;++i) draw(ttout[i-1], yyout[i-1][1], 0.0);

color(MAGENTA);
move(yyout[0][0], yyout[0][1], 0.0);
for (i=2;i<=*npoints;++i) draw(yyout[i-1][0], yyout[i-1][1], 0.0);
*/

/*plot the velocity profile*/
cell-id = winpar.wingid[4-1];
```

```
winset(cell_id);
linewidth(2);
frontbuffer(1);

/*plot the integrated function*/
color(WHITE);
move(ttout[0], yyout[0][1], 0.0);
for  (i=2;i<=*npoints;++i) draw(ttout[i-1], yyout[i-1][1],0.0);
linewidth(1);



/*saving distance, velocity, alpha and alpha-s to data files for plotting */
fout = fopen("alpha_sim.mat","w");
for (i=l;i<=*npoints;++i) fprintf(fout,"%f\n",yyout[i-1][2]);
fclose(fout);

fout = fopen("sd_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",yyout[i-1][1]);
fclose(fout);

fout = fopen("s_sim.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",ttout[i-1]);
fclose(fout);
}
```

```
/*this program uses the runge kutta method to integrate a differential equation
 */
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define abs(x) ( (x) >= 0.0 ? (x): -(x) )
#define sqr(x) ((x)*(x))
#define NP 1000 /* NP = no of points along the path*/
#define SD 2     /*state dimension*/
#define PI              3.141592654

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;


runge_3(t0,tfinal,y0,tol,trace,  ttout,yyout,npoints,ier)
float t0,tfinal,y0[SD],tol;
int trace,*npoints,*ier;
float ttout[NP], yyout[NP][SD];
{
FILE *fout;

float  pow,quant;
int nargin;
float t;
float k1[SD],k2[SD],k3[SD],k4[SD] ,k[SD],vec[SD];
float hmax, hmin, h;
float y[SD], tau, delta;
int icount,i;
long  cell-id; /*cell id*/

/*functions called from outside */
/*float norm(), infnorm(), power-(), min2();*/

*ier = 0;

/*check*/
if(trace){
   printf("in ode23\n");
   printf("t0 = %f, tfinal = %f\n",t0,tfinal);
   printf("y0 = %f,   %f\n",y0[0],y0[1]);
   printf("tol=%f\n",tol);
   printf("trace = %d\n",trace);
         }

/*initialization*/
t = to;
hmax = (tfinal-t)/5.0;
hmin = (tfinal-t)/20000.0;
h = (tf inal-t)/500;
if(trace) printf("at start, h= %f\n",h);
y[0] = y0[0]; y[1] = y0[1];
if(trace) printf("at start, y= %f, %f\n",y[0],y[1]);
icount = 1;
ttout[icount-1] = t;
yyout[icount-1][0] = y[0]; yyout[icount-1][1] = y[1];

/*main loop*/
```

```
while( t<tfinal )
{
  if(t+h>tfinal){ h = tfinal -t; }

  /*slopes*/
  /*  kl = h*f(to,yo) */
  state_alpha_kin(t,y,  k1);
  for (i=1;i<=SD;++i) k1[i-1] = h*k1[i-1];

  /*  k2  =  h*f(to+0.5*h,yo+1/2*k2) */
  for(i=1;i<=SD;++i) vec[i-1] = y[i-1] + 0.5*k1[i-1];
  state_alpha_kin(t+0.5*h, vec,  k2);
  for (i=1;i<=SD;++i) k2[i-1] = h*k2[i-1];

  /* k3 = h*f(to+0.5*h, y0+1/2*k2)   */
  for(i=1;i<=SD;++i) vec[i-1] = y[i-1] + 0.5*k2[i-1];
  state_alpha_kin(t+0.5*h, vec, k3);
  for (i=1;i<=SD;++i) k3[i-1] = h*k3[i-1];

  /* k4 = h*f(to+h, yo +k3)          */
  for(i=1;i<=SD;++i) vec[i-1] = y[i-1] + k3[i-1];
  state_alpha_kin(t+h, vec, k4);
  for (i=1;i<=SD;++i) k4[i-1] = h*k4[i-1];

  for(i=1;i<=SD;++i) k[i-1] = (k1[i-1]+2*k2[i-1]+2*k3[i-1]+k4[i-1])/6.0;

  /*update y */
  for(i=1;i<=SD;++i) y[i-1] = y[i-1] + k[i-1];
  t = t+h;
  icount = icount +1;
  if(trace) printf("icount = %d\n", icount);
  ttout[icount-1] = t;
  for(i=1;i<=SD;++i) yyout[icount-1][i-1] = y[i-1];
  if(trace) printf("y = %f, %f, t = %f\n", y[0], y[1], t);

}

if( t< tfinal ) {printf("singularity likely\n"); *ier =1;}
*npoints = icount;

/*
printf("before drawing\n");
getchar();
*/
cell-id = winpar.wingid[6-1];
winset(cell_id);
linewidth(1);
frontbuffer(1);

/*plot the integrated function*/
color(GREEN);
move(ttout[0], -yyout[0][0], 0.0);
for (i=2;i<=*npoints;++i) draw(ttout[i-11, -yyout[i-l][0], 0.0);

/*
color(YELLOW);
move(ttout[0],yyout[0][1], 0.0);
for (i=2;i<=*npoints;++i) draw(ttout[i-1], yyout[i-1][1], 0.0);

color(MAGENTA);
move(yyout[0][0], yyout[0][1], 0.0);
for (i=2;i<=*npoints;++i) draw(yyout[i-1][0], yyout[i-1][1], 0.0);
*/
```

```
/*saving distance,  alpha and alpha-s to data files for plotting */
fout = fopen("alpha_kin.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",-yyout[i-1][0]*180/PI);
fclose(fout);

fout = fopen("beta_kin.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",atan(2*tan(-yyout[i-1][0]))*180/PI );
fclose(fout);

fout = fopen("s_kin.mat","w");
for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",ttout[i-1]);
fclose(fout);
}
```

```
/*this program uses the runge kutta method to integrate a differential equation
 */
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define abs(x) ( (x) >= 0.0 ?  (x): -(x) )
#define sqr(x) ((x)*(x))
#define NP 1000 /* NP = no of points along the path*/
#define SD 2     /*state dimension*/
#define  PI             3.141592654

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;

extern int store-flag-for-runge-2;

runge_2(t0,tfinal,y0,tol,trace,  ttout,yyout,npoints,ier)
float t0,tfinal,y0[SD],tol;
int trace,*npoints,*ier;
float ttout[NP], yyout[NPl [SD];
{
FILE *fout;

float pow,quant;
int nargin;
float t;
float k1[SD],k2[SD],k3[SD],k4[SD],k[SD], vec[SD];
float hmax, hmin, h;
float y[SD], tau, delta;
int icount,i;
long  cell-id; /*cell id*/

/*functions called from outside */
/*float norm(), infnorm(), power-(), min2();*/

*ier = 0;

/*check*/
if(trace){
   printf("in ode23\n");
   printf("t0 = %f, tfinal = %f\n",t0,tfinal);
   printf("y0 = %f,   %f\n",y0[0],y0[1]);
   printf("tol=%f\n",tol);
   printf("trace = %d\n",trace);
         }

/*initialization*/
t = to;
hmax = (tfinal-t)/5.0;
hmin = (tfinal-t)/20000.0;
h = (tfinal-t)/500;
if(trace) printf("at start, h= %f\n",h);
y[0] = y0[0]; y[1] = y0[1];
if(trace) printf("at start, y= %f, %f\n",y[0],y[1]);
icount = 1;
ttout[icount-1] = t;
yyout[icount-1][0] = y[0]; yyout[icount-1][1] = y[1];

/*main  loop*/
```

```
while( t<tfinal  )
{
  if(t+h>tfinal){ h = tfinal -t; }

  /*slopes*/
  /*  kl = h*f(to,yo) */
  state_alpha(t,y,  k1);
  for (i=1;i<=SD;++i) k1[i-1] = h*k1[i-1];

  /*  k2  = h*f(to+0.5*h,yo+1/2*k2) */
  for(i=1;i<=SD;++i) vec[i-1] = y[i-1] + 0.5*k1[i-1];
  state_alpha(t+0.5*h, vec,  k2);
  for (i=1;i<=SD;++i) k2[i-1] = h*k2[i-1];

  /* k3 = h*f(to+0.5*h, y0+1/2*k2)   */
  for(i=l;i<=SD;++i)  vec[i-1] = y[i-1] + 0.5*k2[i-1];
  state_alpha(t+0.5*h, vec, k3);
  for (i=1;i<=SD;++i) k3[i-1] = h*k3[i-1];

  /* k4 = h*f(to+h, yo +k3)          */
  for(i=l;i<=SD;++i)  vec[i-1] = y[i-1] + k3[i-1];
  state_alpha(t+h,  vec, k4);
  for  (i=1;i<=SD;++i) k4[i-1] = h*k4[i-1];

  for(i=1;i<=SD;++i) k[i-1] = (k1[i-1]+2*k2[i-1]+2*k3[i-1]+k4[i-1])/6.0;

  /*update y */
  for(i=l;i<=SD;++i)  y[i-1] = y[i-1] + k[i-1];
  t = t+h;
  icount = icount +1;
  if(trace) printf("icount = %d\n", icount);
  ttout[icount-1] = t;
  for(i=1;i<=SD;++i) yyout[icount-1] [i-l] = y[i-1];
  if(trace) printf("y = %f, %f,  t = %f\n", y[0], y[1], t);

}

if( t< tfinal ) {printf("singularity likely\n"); *ier =1;}
*npoints = icount;

/*
printf("before drawing\n");
getchar();
*/
cell-id = winpar.wingid[6-1];
winset(cell_id);
linewidth(1);

color(BLACK);
clear();

/*plot the integrated function*/
color(RED);
move(ttout[0], -yyout[0][0], 0.0);
for (i=2;i<=*npoints;++i) draw(ttout[i-1], -yyout[i-1][0], 0.0);

color(YELLOW);
move(ttout[0],-yyout[0][1], 0.0);
for (i=2;i<=*npoints;++i) draw(ttout[i-1], -yyout[i-1][1], 0.0);

/*
color(MAGENTA);
move(yyout[0][0], yyout[0][1], 0.0);
for (i=2;i<=*npoints;++i) draw(yyout[i-1][0], yyout[i-1][1], 0.0):
*/
```

```
swapbuffers();


/*saving distance, alpha and alpha-s to data files for plotting */
if (store_flag_for_runge_2==1)
{
  fout = fopen("alpha_out.mat","w");
  for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",-yyout[i-1][0]*180.0/PI);
  fclose(fout);

  fout = fopen("alphas_out.mat","w");
  for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",-yyout[i-1][1]*180.0/PI);
  fclose(fout);

  fout = fopen ("s-out .mat","w");
  for (i=1;i<=*npoints;++i) fprintf(fout,"%f\n",ttout[i-1]);
  fclose(fout);
}

}
```

```
/*this program uses the runge kutta method to integrate a differential equation
 *
 */
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define abs(x) ( (x) >= 0.0 ? (x): -(x) )
#define sqr(x) ((x)*(x))
#define NP 1000 /* NP = no of points along the path*/
#define SD 2     /*state dimension*/

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;


runge(t0,tfinal,y0,tol,trace,   tout,yout,npoints,ier)
float t0,tfinal,y0[SD],tol;
int trace,*npoints,*ier;
float tout[NP], yout[NP][SD];
{
float pow,quant;
int nargin;
float t;
float k1[SD],k2[SD],k3[SD],k4[SD],k[SD], vec[SD];
float hmax, hmin, h;
float y[SD], tau, delta;
int icount,i;
long  cell-id; /*cell id*/

/*functions called from outside */
/*float norm(), infnorm(), power-(), min2();*/

*ier = 0;

/*check*/
if(trace){
   printf("in ode23\n");
   printf("t0 = %f, tfinal = %f\n",t0,tfinal);
   printf("y0 = %f,   %f\n",y0[0],y0[1]);
   printf("tol=%f\n",tol);
   printf("trace = %d\n",trace);
         }

/*initialization*/
t = to;
hmax = (tfinal-t)/5.0;
hmin = (tfinal-t)/20000.0;
h = (tfinal-t)/500;
if(trace) printf("at start, h= %f\n",h);
y[0] = y0[0]; y[1] = y0[1];
if(trace) printf("at start, y= %f, %f\n",y[0],y[1]);
icount = 1;
tout[icount-1] = t;
yout[icount-1][0] = y[0]; yout[icount-1][1] = y[1];

/*main loop*/
while( t<tfinal  )
{
  if(t+h>tfinal){ h = tfinal -t; }

  /*slopes*/
```

```
  /*  kl  =  h*f(to,yo) */
  stateq(t, y,  kl):
  for (i=l;i<=SD;++i) kl[i-1] = h*kl [i-1];

  /*  k2  =  h*f(to+0.5*h,yo+1/2*k2) */
  for(i=l;i<=SD;++i)  vec[i-1] = y[i-1] + 0.5*kl[i-1];
  stateq(t+0.5*h, vec,  k2);
  for (i=1; i<=SD;++i) k2 [i-l] = h*k2[i-1];

  /* k3 = h*f(to+0.5*h, y0+1/2*k2)    */
  for(i=l;i<=SD;++i)  vec[i-1] = y[i-1] + 0.5*k2[i-1];
  stateq(t+0.5*h, vec, k3);
  for  (i=1;i<=SD;++i) k3[i-1] = h*k3[i-1];

  /* k4 = h*f(to+h, yo +k3)           */
  for(i=1;i<=SD;++i) vec[i-1] = y[i-1] + k3[i-1];
  stateq(t+h, vec, k4);
  for  (i=1;i<=SD;++i) k4[i-1] = h*k4[i-1];

  for(i=l;i<=SD;++i)  k[i-1] = (k1[i-1]+2*k2[i-1]+2*k3[i-1]+k4[i-1])/6.0;

  /*update y */
  for(i=1;i<=SD;++i) y[i-1] = y[i-1] + k[i-1];
  t = t+h;
  icount = icount +1;
  if(trace) printf("icount = %d\n", icount);
  tout[icount-1] = t;
  for(i=1;i<=SD;++i) yout[icount-1] [i-l] = y[i-1];
  if(trace) printf("y = %f, %f,  t = %f\n", y[0], y[1], t);

}

if( t< tfinal ) {printf("singularity likely\n"); *ier =1;}
*npoints = icount;

cell-id = winpar.wingid[5-1];
winset(cell_id);
linewidth (1);

color(BLACK);
clear();

/*plot the integrated function*/
color(RED);
move(tout[0], yout[0][0], 0.0);
for (i=2;i<=*npoints;++i) draw(tout[i-1], yout[i-1][0], 0.0);

color(YELLOW);
move(tout[0],yout[0][1], 0.0);
for (i=2;i<=*npoints;++i) draw(tout[i-1], yout[i-1][1], 0.0);

color(MAGENTA);
move(yout[0][0], yout[0][1], 0.0);
for (i=2;i<=*npoints;++i) draw(yout[i-1][0], yout[i-1][1], 0.0);

swapbuffers();
}
```

```
/*****************************************************************************
written by Satish
program to optimize the velocity profile along a specified path
*****************************************************************************/

#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>

#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define PI              3.141592654

#define MAX-NPAR 30 /*the number of parameters for optimization*/

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;

float current_min_time; /*to store the best time*/
char QTY[3]; /*to decide which to optimize*/

float best-traj-time; /*stores the time for the best trajectory*/

extern int current_int_increment;/*current point increment*/

opt-trajectory(qty)
char qty[3];
{

float par[MAX_NPAR],cost,costmin,phi[MAX_NPAR],tht[MAX_NPAR],xflg[MAX_NPAR];
int npar,ipt,ipr,icn,npiter,itlim ;

float par2[MAX_NPAR],cost2,costmin2,phi2[MAX_NPAR],tht2[MAX_NPAR],xflg2[MAX_NPAR];
int npar2,ipt2,ipr2,icn2,npiter2,itlim2 ;

extern get-time-();
extern get-opt_time_along_path_();
float dels,dlmin ;
float dels2,dlmin2 ;

int i:

/*for coarse computation of control along the path*/
current-int-increment = 7;  /*7;*/ /*1;*/ /*5;*/ /*10;*/

/*check whether path and/or vel. profile should be optimized*/
for (i=1;i<=3;++i) QTY[i-1] = qty[i-1];

if (
     ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='h') )
                        ||
     ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='2') )
                        ||
     ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='3') )
                        ||
     ( (QTY[0]=='v') && (QTY[1]=='e') && (QTY[2]=='l') )
                        ||
     (  (QTY[0]=='e') && (QTY[1]=='n') &&(QTY[2]=='v'))
                        ||
```

```
    (  (QTY[0]=='T') && (QTY[1]=='R') && (QTY[2]=='J'))
                        ||
    ( (QTY[0]=='T') && (QTY[1]=='R') && (QTY[2]=='2') )
                        ||
    ( (QTY[0]=='c') && (QTY[1]== 'V') && (QTY[2]=='l') )
  )
{
  /* initialise the parameters of optimisation */

  cost = 0.0 ;

  ipt =0 ;
  ipr = 4;
  icn = 4 ;
  npiter=l ;
  dels = 1.0 ;

  dlmin = 0.001 ;
  itlim = 1000;


  /*convert the control points of the acceleration profile to parameters*/
  btopar(&npar,par);

  /*initialize the best time at the moment*/
  current-min-time = 1000.0;

  /*iterate to get the optimum value of the link lengths*/
  patsh_(par,&cost,phi,tht,xflg,&npar,&dels,&dlmin,&itlim,&ipt,&npiter,&ipr,&icn,ge

  /*convert the optimal parameters to control points of the velocity profile*/
  partob(npar,par);
}
else /*if QTY[] = "trj"*/
{
  /* initialise the parameters of optimisation */

  cost2 = 0.0 ;

  ipt2 =0 ;
  ipr2 = 4;
  icn2 = 4 ;
  npiter2=1 ;
  dels2 = 1.0 ;
  dlmin2 = 0.001 ;
  itlim2 = 1000;


  /*convert the control points of path to parameters*/
  QTY[O] = 'p'; QTY[1] = 't'; QTY[2] = 'h';
  btopar(&npar2,par2);

  /*initialize the best time at the moment*/
  best-traj-time = 1000.0;

  /*iterate to get the optimum value of the link lengths*/
  patsh_(par2,&cost2,phi2,tht2,xflg2,&npar2,&dels2,&dlmin2,&itlim2,&ipt2,&npiter2,&

  /*convert the optimal parameters to control points of path*/
  QTY[0] = 'p'; QTY[1] = 't'; QTY[2] = 'h';
  partob(npar2,par2);
}


}
```

```
/*****************************************************************************
written by Satish
computes the limitc curve for the vehicle problem
*****************************************************************************/
#include  <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "limitc.h"
#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define PI              3.141592654

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;

extern float s[500],x[500][2],xs[500][2],xss[500][2];
extern float velpro_s[1000], velpro_sd[1000], velpro_sdd[1000];
extern int PATH_NO;
extern int VEL_PATH_NO;
extern struct vehicle {float mass, Ic, d, alpha, alpha-f, alpha_r, Ft, Beta,
                       fnr, fnf, width, wheel-length, wheel_width; } car;
extern struct paths (float theta[500],theta_s[500],theta_ss[500];} path;

/*****************************************************************************/
/* compute-limit-curve0 computes the velocity limit curve                    */
/*****************************************************************************/
float compute-limit-curve0
{
}

/*****************************************************************************/
/*  vel_limit_at_state() finds the velocity limit for given s, sd and        */
/*                       alpha                                               */
/*****************************************************************************/
float vel_limit_at_state(dis, theta-dis, thetas-dis, alpha_dis, alphas_dis)
float dis, theta_dis, thetas_dis,  alpha_dis,  alphas_dis;
{
 float anglex();
 float f_fn();
 float f_alpha_r();

 float sd_at_dis, alpha-dis, alphas_dis;
 float sdlim_at_dis;
 float kappa_at_dis;
 float sign of-kappa;
 float driving-force, steering-angle;
 float driving-force-limit, steering-angle-limit;

 float alpha-front, alpha-rear;
 float Crear, Cfront;  /*force constant on rear wheel*/
 float  psi-front;

 Crear  = 80000;
 Cfront = -80000;

 kappa-at-dis = thetas-dis;
 if     (kappa_at_dis> 0.0001)   sign-of-kappa = 1.0;
 else if(kappa_at_dis< -0.0001)  sign-of-kappa = -1.0;
 else                            return(lOOO.0);  /*infinite velocity limit*/
```

```
 /*initialize*/
 sdlim_at_dis = 0.0;

 /*optimize over alpha-dis, alphas-dis and beta*/
 for(alpha_dis= -0.5*PI; alpha-dis <= 0.5*PI; alpha_dis=alpha_dis+0.01)
 {
  for(alphas_dis= -0.5*PI; alphas-dis <= 0.5*PI; alphas_dis=alphas_dis+0.01)
  {
    for(steering_angle= -0.5*PI; steering-angle <= 0.5*PI;
                                  steering_angle=steering_angle+0.05)
    {
       /*determine  alpha-front*/
       psi-front  = anglex( cos(alpha_dis),
                            car.d*(thetas_dis+alphas_dis) - sin(alpha_dis) );

       alpha_front = steering-angle - psi-front;

       /*determine  alpha-rear*/
       alpha_rear = f_alpha_r(dis, 1.0, alpha-dis, alphas-dis, thetas-dis);

       sd_at_dis = (
                      sign-of-kappa*abs( driving-force-limit*sin(alpha_dis) )
                    + f_fn(alpha_front,Cfront)*cos(alpha_dis+steering_angle)
                    + f_fn(alpha_rear,Crear)*cos(alpha_dis)
                    )/car.mass/kappa_at_dis;
       if (sd_at_dis>sdlim_at_dis) sdlim_at_dis = sd_at_dis;
    }
  }
 }
}
```

```
/*****************************************************************************
written by Satish
computes the dynamics of the vehicle
*****************************************************************************/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "runge.h"

#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define  PI             3.141592654

extern struct p [int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;

extern float s[500],x[500][2],xs[500][2],xss[500][2];
extern float velpro_s[1000], velpro_sd[1000], velpro_sdd[1000];
extern struct tv {int nconv;  float conv[20][2];} bzvel_;
extern struct tw {int nconp;  float conp[20][2];} bzw_;
extern int VEL_PATH_NO;
extern int PATH NO;
extern struct paths {float theta[500],theta_s[500],theta_ss[500];} path;
extern struct forces {float beta[1000], drive[1000], dis[1000]; int np;} force;

extern int  store-flag-for-runge _2;

int ier_vel, ier_path;

extern float current_min_time;

extern int path-type;

extern char QTY[3]; /*to decide what to optimize*/

extern float  slope_0_desired,  slope_f_desired; /*desired initial and final
                                                   theta's */

int existence-flag; /* 0 if roots exists, 1 if they do not*/

int violation-of-wheel-slip;  /* 0 if wheel slip angle limits are not violated,
                                 1 if they are*/

int violation_of_fr_ellipse; /* 0 if friction ellipse is not violated,
                                 1 if it is*/

extern int current_int_increment:         /*current point increment*/
extern float constant_velocity_of_car;    /*initial velocity*/


get-time-(npar,par,Cost)
int *npar;
float *par,*cost ;

{
 FILE *fout;

 float integrate_velocity_profile();

 int  i;
```

```
int   ier;
int   penalty-flag-beta,  penalty-flag-drive;

float penalty-beta, penalty-drive;
float penalty-slope-O,  penalty-slope-f;  /*penalty to ensure that the
                                            slopes at the initial and final
                                            points are fixed*/
float penalty_for_no_roots; /*penalty for non-existence of roots*/
float penalty-for-violation-of-wheel-slip:
                            /*penalty for violation of wheel slip angle limits*/
float penalty_for_violation_of_fr_ellipse;
                           /*penalty for violation of friction ellipse*/
float C-slope-O, C-slope-f;  /*weighting factors for the penalty on the
                               initial and final slopes*/


C-slope-0 = 10.0;
C-slope-f = 10.0;

/*convert the parameters to control points of the velocity profile*/
partob(*npar,par);
/*if(bzvel_.conv[0][1] > 15.0) bzvel_.conv[0][1] = 15.0;*/

if(
   ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='h') )
                          II
   ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='2')  )
                          ||
   ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='3')  )
                          ||
   ( (QTY[0]=='T') && (QTY[1]=='R') && (QTY[2]=='J')  )
                          ||
   ( (QTY[O]=='T') && (QTY[1]=='R') && (QTY[2]=='2') )
  )
     {generate_path_profile(path_type); drw_path_w(1); drw_theta(2);}

/*generate the velocity profile using the bspline*/
generate_velocity_profile();

/*penalty for incorrect bspline*/
if ( (ier_vel == 1) I |(ier_path==1) ) { *cost = 1000.0; return;)


/*compute the time taken by integrating ds/sd along the velocity profile*/
*cost = integrate_velocity_profile();

/*
printf("cost before penalty on initial velocity = %f\n", *cost):
if(velpro_sd[0] > constant-velocity-of-car) *cost = *cost + 15.0;
printf("cost after penalty on initial velocity = %f\n", *cost);
*/

printf("cost before penalty on existence of solution = %f\n", *cost);

/*compute alpha and beta and driving force along the path to check if
  constrants are violated*/
store_flag_for_runge_2 = 0;    /*no storing data files*/
trace2 = Of
runge_2(t02,tfinal2,x02,tol2,trace2,  tout2,yout2,&npoints2,&ier);

find_path_steer_angles(current_int_increment);

/*penalty for nonexistence of roots*/
if (existence_flag==1) penalty_for_no_roots = 5.0;
else                   penalty_for_no_roots = 0.0;
```

```
*cost = *cost + penalty_for_no-roots;
printf("cost after penalty on existence of solution = %f\n", *cost);

/*penalty for violation of wheel slip angle limits*/
printf("cost before penalty on wheel slip violation = %f\n",*cost);
if (violation-of-wheel-slip==l) penalty-for-violation-of-wheel-slip = 5.0;
else                            penalty-for-violation-of-wheel-slip = 0.0;
*cost = *cost + penalty-for-violation-of-wheel-slip;
printf("cost after penalty on wheel slip violation = %f\n",*cost);

/*penalty for violation of friction ellipse*/
printf("cost before penalty on friction ellipse = %f\n", *cost);
if (violation-of_fr_ellipse==1) penalty_for_violation of fr ellipse = 5.0;
else                            penalty_for_violation_of_fr_ellipse = 0.0;
*cost = *cost + penalty_for_violation_of_fr_ellipse;
printf("cost after penalty on friction ellipse = %f\n",*cost);


plot_steer_and_drive_forces(current_int_increment,7);

/*compute the penalties due to the violation of forces*/
penalty-beta       = 0.0;
penalty-drive      = 0.0;
penalty-flag-beta  = 0;
penalty_flag-drive = 0;
/*for (i=1;i<=force.np;++i)*/
for (i=1;i<=force.np;i=i+current_int_increment)
{
  /*if ( abs(force.beta[i-1]) > 20.0*PI/180.0 ) */
  if ( abs(force.beta[i-1]) > 50.0*PI/180.0 )
      { penalty-beta = penalty-beta + 1.0; penalty-flag-beta = 1;}
  /*
  if ( abs(force.drive[i-1]) > 0.3*1550*9.81 )
      { penalty-drive = penalty-drive + 1.0; penalty-flag-drive = 1;}
      */

  /*if positive ft exceeds 2000 or negative ft is less than -6000
    impose penalty                                                */
  if (
       ( force.drive[i-1] >   3000.0 ) /*old limit 2000*/
                          ||
       ( force.drive[i-1] < -6000.0 )
     )
      { penalty-drive = penalty-drive + 1.0; penalty-flag-drive = 1;}
}

if (penalty-flag-beta==l) printf("beta constraints violated\n");
if (penalty-flag-drive==l) printf("drive constraints violated\n");

printf("cost before penalty on control = %f\n", *cost):

*cost = *cost + penalty_beta + penalty_drive;

printf("cost after penalty on control = %f\n", *cost);

/*compute the penalty on the slopes at the end points*/
if(
   (  (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='h') )
                          ||
   (  (QTY[0]=='T') && (QTY[1]=='R') && (QTY[2]=='J') )
  )
     {
      penalty-slope-0 = C_slope_0*abs(path.theta[0]         - slope_0_desired);
      penalty_slope-f = C_slope_f*abs(path.theta[PATH_NO-1] - slope_f_desired);
      *cost = *cost + penalty-slope-0 + penalty-slope-f;
      printf("cost after penalty on slope = %f\n",*cost);
```

```
 if ( *cost < current min time ) /*then store the current parameters*/
 I
   current_min_time = *cost:

   fout = fopen("vel_profile.opt","w");
    fprintf(fout,"%d\n",bzvel_.nconv);
    for(i=1;i<=bzvel_.nconv;++i)fprintf(fout,"%f %f ",
                                bzvel_.conv[i-1][0],bzvel_.conv[i-1][1]);
   fclose(fout);

   fout = fopen("bw_space.opt","w");
    fprintf(fout,"%d\n",bzw_.nconp);
    for(i=1;i<=bzw_.nconp;++i)fprintf(fout,"%f %f ",
                                bzw_.conp[i-1][0],bzw_.conp[i-l][1]);
   fclose(fout);

   fout = fopen("constant_velocity_of_car.opt","w");
    fprintf(fout, "%f\n",constant_velocity_of_car);
   fclose(fout);

   /*store the force and steering angles for plotting*/
   /***
   fout = fopen("steer.opt","w");
    for (i=1;i<=force.np;++i) fprintf(fout,"%f\n",force.beta[i-1]);
   fclose(fout);

   fout = fopen("drive.opt","w");
    for (i=1;i<=force.np;++i) fprintf(fout,"%f\n",force.drive[i-1]);
   fclose(fout);

   fout = fopen("pdis.opt","w");
    for (i=1;i<=force.np;++i) fprintf(fout, "%f\n",force.dis[i-1]);
   fclose (fout);
   ***/
 1


}


float integrate_velocity_profile()
{
 float time_taken;
 float penalty_for_low_vel;

 int  i;

 time_taken = 0.0; /*initialize*/

 for (i=1;i<=VEL_PATH-NO-l; ++i)
 {
  /*
  if (abs(velpro_sd[i-1]) < 3.0) penalty-for-low-vel = 1000.0;
  */
  if (abs(velpro_sd[i-1]) < 2.5) penalty-for-low-vel = 1000.0;
  else                           penalty_for_low_vel = 0.0:

  if (velpro_sd[i-1]<0.01)  velpro_sd[i-1] = 0.01;

  time-taken = time-taken + (velpro_s[i]-velpro_s[i-1])/velpro_sd[i-l]
                          + penalty_for_low_vel;;
 }

 return(time_taken);
```

```
/*************************************************************************
written by Satish
computes the optimal time along a given path to be used as the cost function
in an optimization to find the optimal path
*************************************************************************/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "runge.h"

#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define PI              3.141592654

extern float s[500],x[500][2],xs[500][2],xss[500][2];
extern int PATH-NO;
extern float current_min_time;
extern float best traj_time;
extern int path-type;

extern struct tw {int nconp; float conp[20][2];} bzw_;
extern struct tv {int nconv;  float conv[20][2];} bzvel_;

extern char QTY[3]; /*to decide what to optimize*/

get_opt_time-along-path-(npar,par,cost)
int *npar;
float *par,*cost ;

{
 FILE *fout;
 int  i;

 /*convert the parameters to control points of the path*/
 QTY[0] = 'p'; QTY[1] = 't'; QTY[2] = 'h':
 partob(*npar,par);

 /*generate the path*/
 generate_path_profile(path_type);
 drw_path_w(1);
 drw_theta(2);

 /*final time for runge-kutta integration*/
 tfina12 = s[PATH_NO-1];

 /*optimize velocity profile for the given path*/
 opt_trajectory("vel");
 *cost = current_min_time;

 if( *cost<best_traj_time )
 {
   best-traj-time = *cost;

   fout = fopen("vel_profile.opt","w");
    fprintf(fout,"%d\n",bzvel_.nconv);
    for(i=1;i<=bzvel_.nconv;++i)fprintf(fout,"%f %f ",
                                bzvel_.conv[i-1][0],bzvel_.conv[i-1][1]);
   fclose(fout);

   fout = fopen("bw_space.opt","w");
    fprintf(fout,"%d\n",bzw_.nconp);
    for(i=1;i<=bzw_.nconp;++i)fprintf(fout,"%f %f ",
                                bzw_.conp[i-1][0],bzw_.conp[i-1][1]);
   fclose(fout);
```

```
/*************************************************************************
LABORATORY FOR ROBOTICS AND AUTOMATION,MANE,UCLA,1994
-------------------------------------------------
written by satish
*************************************************************************/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define  abs(x)          (((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define  PI             3.141592654

extern struct p (int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                  } winpar ;
extern struct t {int nconp; float conp[20][2];} bz_;
extern struct tw [int nconp;  float conp[20][2];} bzw_;
extern struct tv {int nconv;  float conv[20][2];} bzvel_; /*velocity profile*/
extern struct u {float angx,angy,dx,dy; int px,py;} gr;

extern float init_point[2];
extern float constant-velocity-of-car;

extern int path-type;
extern int vel_type;

gclpt_w(cell_no)
int cell_no:
{
 long cell-id; /*cell id*/
 int j,n_point;
 float x_max,x_min,y_max,y_min; /*window coords*/
 float cur_x,cur_ y; 7*cursor window coords*/
 float dist, dist_min;
 float  pos[4];

 int type-flag; /*to change initial velocity and acceleration profile*/

 type-flag = 0;

 /************************
  choose the window
  ************************/
 cell_id = winpar.wingid[cell_no-1]  ;
 winset(cell_id);

 x_min = winpar.wincoord[0][cell_no-1] ;
 x_max = winpar.wincoord[1][cell_no-1] ;
 y_min = winpar.wincoord[2][cell_no-1] ;
 y_max = winpar.wincoord[3][cell_no-1] ;
 pos[0] = winpar.winposi[0][cell_no-1] ;
 pos[1] = winpar.winposi[1][cell_no-1] ;
 pos[2] = winpar.winposi[2][cell_no-1] ;
 pos[3] = winpar.winposi[3][cell_no-1] ;

/*************************************************************
convert cursor screen coordinates to window coordinates
*************************************************************/
 cur-x = x_min + (x_max-x_min)*(gr.px-pos[0])/(pos[1]-pos[0]);
 cur_y = y_min + (y_max-y_min)*(gr.py-pos[2])/(pos[3]-pos[2]);
```

```
/*************************************************************************
if cell_no = 2,  get the nearest control point
*************************************************************************/
if (cell-no == 2)
{
 dist_min = 1000000.0;

 for(j=1;j<=bz_.nconp;++j)
 {
    dist = sqrt(sqr(cur_x-bz_.conp[j-1][0])+sqr(cur_y-bz_.conp[j-1][1]));
    if(dist<dist_min)
    {
       dist min = dist;
       n-point = j;
    }
 }


 if(n_point != 1) bz_.conp[n_point-1][0] = cur_x;
 bz_.conp[n_point-1][1] = cur-y;
}
/*******r*****************************************************************
    if cell_no = 1,  change the value of the initial point
*************************************************************************/
else if (cell_no==1)
{
  if (path-type==0)
  {
    dist = sqrt(sqr(cur_x-init_point[0])+sqr(cur_y-init_point[1]));
    if (dist < 100)
    {
     init_point[0] = cur-x;
     init_point[1] = cur-y;
    }
  }
  else
  {
    dist_min = 1000000.0;
    for(j=1;j<=bzw_.nconp;++j)
    {
       dist = sqrt(sqr(cur_x-bzw_.conp[j-1][0])+sqr(cur_y-bzw_.conp[j-1][1]));
       if (dist<dist_min)
       {
          dist_min = dist;
          n_point = j;
       }
    }
    bzw_.conp[n_point-1][0] = cur-x;
    bzw_.conp[n_point-1][1] = cur-y;

  }

}
/*************************************************************************
if cell_no = 4,  get the nearest control point for the velocity profile
*************************************************************************/
else if (cell-no == 4)
{
 dist_min = 1000000.0;

 for(j=1;j<=bzvel_.nconv;++j)
 {
    dist = sqrt(sqr(cur_x-bzvel_.conv[j-1][0])+sqr(cur_y-bzvel_.conv[ j-11 [1]));
    if(dist<dist_min)
    {
       dist min = dist;
```

```
        n-point = j;
        type-flag = 0;
      }
  }

  if (vel_type==0)
  {
    dist=sqrt(sqr(cur_x-bzvel_.conv[0][0])+sqr(cur_y-constant_velocity_of_car));
    if(dist<dist_min)
    {
       dist_min = dist;
       type-flag = 1;
    }

    if(type_flag==0)
    I
      if(n_point != 1) bzvel_.conv[n_point-1][0] = cur_x;
      bzvel_.conv[n_point-1][1] = cur-y;
    }
    else
      constant-velocity-of-car = cur_y;

  }
  else
  {
      if(n_point != 1) bzvel_.conv[n_point-1][0] = cur-x;
      bzvel_.conv[n_point-1][1] = cur-y;
  }
}
else
{
}

}
```

```
/*************************t************************************************
written by Satish
computes the dynamics of the vehicle
*****************************************************************************/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define  PI             3.141592654

/***********critical slip angles for the front and rear tires*****************/
#define  ALPHA_F_CR             10*PI/180 /* 10 degrees */
#define  ALPHA_R_CR             10*PI/180 /* 10 degrees */
/*****************************************************************************/

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                           } winpar ;

extern float s[500],x[500][2],xs[500][2],xss[500][2];
extern float vel_s[1000],vel_x[1000][2],vel_xs[1000][2],vel xss[1000][2];
                 /*for the velocity profile alogn the path*/
extern float accel_s[1000]; /*the acceleration sdd(s) along the path*/
extern float velpro_s[1000], velpro_sd[1000], velpro_sdd[1000];
extern struct t {int nconp; float conp[20][2];} bz_;
extern struct tw {int nconp; float conp[20][2];} bzw_;
extern struct tv {int nconv;  float conv[20][2];} bzvel_;
extern int PATH_NO;
extern int VEL_PATH_NO;
extern struct vehicle {float mass, Ic, d, alpha, alpha-f, alpha_r, Ft, Beta,
                       fnr,  fnf,  width,  wheel-length,  wheel_width;} car;
extern struct paths {float theta[500],theta_s[500],theta_ss[500];} path;

/*for the bspline representing the path slope*/
extern float sangle[500],xangle[500][2],xangles[500][2],xangless[500][2];
extern float init_point[2];

/*actual value of theta to check for accuracy of the differentiation*/
extern float theta_actual[500];

/*constant velocity of car*/
extern float constant-velocity-of-car;

/*flag for the specified path optimization*/
extern int ier_vel, ier_path;

/*flag for path representation*/
extern int path type;
/*flag for velocity representation*/
extern int vel_type;

/*store profile at switch*/
extern float vel_profile_at_switch[400][2];

/*****************************************************************************/
/*      f_alpha_r() returns alpha_r -> slip angle for the rear  wheel      */
/*****************************************************************************/
float f_alpha-r(dis, sd-dis, alpha-dis, alphad-dis, thetad-dis)
float dis, sd-dis, alpha-dis, alphad-dis, thetad-dis:
{
```

```
float anglex();

return( anglex( sd_dis*cos(alpha_dis),-(sd_dis*sin(alpha_dis)+car.d*(alphad_dis+the
}

/*************************************************************************/
/*      f_alpha_r2() returns alpha_r -> slip angle for the rear  wheel      */
/*************************************************************************/
float f_alpha_r2(dis, sd-dis, alpha-dis, alphas-dis, thetas-dis)
float dis, sd-dis, alpha-dis, alphas-dis, thetas-dis;
{
float anglex();

return( anglex( cos(alpha_dis),-(sin(alpha_dis)+car.d*(alphas_dis+thetas_dis) )) );
}


/*************************************************************************/
/*      f_alpha_f() returns alpha_f -> slip angle for the front wheel      */
/*************************************************************************/
float f_alpha_f(dis, sd-dis, alpha-dis, alphad-dis, thetad-dis, beta-dis)
float dis, sd_dis, alpha-dis, alphad-dis, thetad-dis, beta-dis;
{
float anglex();

return( beta-dis - anglex( sd_dis*cos(alpha_dis),-sd_dis*sin(alpha_dis)+car.d*(alph
}

/*************************************************************************/
/*      f_fn() returns the normal force acting on a wheel                  */
/*************************************************************************/
float f_fn(alpha_dis, C)
float alpha-dis, C; /* C is the force constant */
{
float sgn-sin-alpha; /* sign of sin(alpha) */

if      (sin(alpha_dis) >  0.0001) sgn-sin-alpha = 1.0;
else if (sin(alpha_dis) < -0.0001) sgn-sin-alpha = -1.0;
else                               sgn-sin-alpha = 0.0;

return( -C*abs(alpha_dis)*sgn_sin_alpha );
}

float f_fn_2(alpha_dis, C) /*implements the saturation function*/
float alpha_dis, C; /* C is the force constant */
{
float sgn-sin-alpha; /* sign of sin(alpha) */

if      (sin(alpha_dis) >  0.0001) sgn-sin-alpha = 1.0;
else if (sin(alpha_dis) < -0.0001) sgn-sin-alpha = -1.0;
else                               sgn-sin-alpha = 0.0;


return(
       ( (   abs(alpha_dis)<ALPHA_F-CR     ) ?
         (-C*abs(alpha_dis)*sgn_sin_alpha)  : (-C*ALPHA_F_CR*sgn_sin_alpha) )
          );
}

/*************************************************************************/
/*      f_alpha_dd() returns d2(alpha(s))/dt2                              */
/*************************************************************************/
float f_alpha-dd(dis, sd_dis, sdd_dis, alpha-dis, alphad-dis, thetad-dis, thetadd_d
float dis, sd_dis, sdd dis, alpha dis, alphad dis, thetad_dis, thetadd_dis;
float Cr; /* Cr is the-force constant for the-rear wheel */
{
```

```
float d2alphadt2;
float alpha-rear;
float  f_kappa(), f_alpha_r(), f_fn();
float kappa-at-s;

kappa-at-s = thetad_dis/sd_dis;

alpha_rear = f_alpha_r(dis, sd-dis, alpha-dis, alphad-dis, thetad-dis) ;

d2alphadt2 = - car.Ic*thetadd_dis
             + car.mass*car.d*(-sdd_dis*sin(alpha_dis) +
             kappa-at_s*sqr(sd_dis)*cos(alpha_dis))
             - 2*car.d*f_fn(alpha_rear, Cr);

d2alphadt2 = d2alphadt2/car.Ic;

return( d2alphadt2 );
}

/*************************************************************************/
/*      f_kappa() returns inverse of the path radius of curvature        */
/*************************************************************************/
float f_kappa(dist)
float dist; /*dist = s = distance along the path*/
{
float get_quantity_at_s();
return(   get_quantity_at_s("ths",dist)   );
}

/*************************************************************************/
/*      f_Beta() returns the control Beta, given the states              */
/*************************************************************************/
float f_Beta(dis, sd dis, sdd_dis, alpha-dis, alphad-dis, thetad-dis, Cr)
float dis, sd_dis, sdd dis, alpha dis, alphad dis, thetad dis;
float Cr; /* Cr is the-force constant for the-rear wheel */
{
float A;
float  alpha-rear;
float f_kappa(), f_fn();

alpha_rear = f_alpha_r(dis, sd-dis, alpha-dis, alphad-dis, thetad-dis) ;

A = car.mass*(-sdd_dis*sin(alpha_dis) +
    f_kappa(dis)*sqr(sd_dis)*cos(alpha_dis)) - f_fn(alpha_rear, Cr);

/* use golden search to find the roots of the equation ?? */
return(l);
}


/*************************************************************************/
/* generate_velocity_profile() generates a velocity profile using B-splines*/
/*************************************************************************/
generate_velocity_profile_2()
{
FILE *fout;
long  cell_id;
int  i,  ier;

int  store-plots;


/*the first and last control points for the x dir must be the same as for the
theta profile all others must lie in between these two*/
bzvel_.conv[0][0] = bz_.conp[0][0];
bzvel_.conv[bzvel_.nconv-1][0] = bz_.conp[bz_.nconp-1][0];
```

```
/*if the x coordinate of any point is greater than that of the last point
 make thevalue that of the last point*/
for (i=2;i<=bzvel_.nconv-1;++i)
if (bzvel_.conv[i-1][0] >= bzvel_.conv[bzvel_.nconv-1][0])
    bzvel_.conv[i-1][0] = bzvel_.conv[bzvel_.nconv-1][0];

/*if the x coordinate of any point is less than that of the first point
 make thevalue that of the first point*/
for (i=2;i<=bzvel_.nconv-1;++i)
if (bzvel_.conv[i-1][0] <= bzvel_.conv[0][0])
    bzvel_.conv[i-1][0] = bzvel_.conv[0][0];

/*generate the velocity profile using bsplines*/
bspline_vel2_(&VEL_PATH_NO,vel_s,vel_x,vel_xs,vel_xss,&ier_vel) ;
printf("vel_path_no = %d\n", VEL_PATH_NO);

/* compute the distance velpro_s[i-1] = vel_x[i-1][0]= maximumx coordinate*/
for (i=1;i<=VEL_PATH_NO;++i) velpro_s[i-1] = vel_x[i-1][0];

/*compute sd along the path*/
for (i=1;i<=VEL_PATH_NO;++i) velpro_sd[i-1]= vel_x[i-1][1];

/*compute sdd using finite differences*/
for (i=1;i<=VEL_PATH_NO-1;++i)
velpro_sdd[i-1]= (sqr(velpro_sd[i])-sqr(velpro_sd[i-1]) )/(velpro_s[i]-velpro_s[i-1]
velpro_sdd[PATH_NO-1] = velpro_sdd[PATH_NO-2];

for (i=2;i<=VEL_PATH-NO-l;++i)
velpro_sdd[i-1]= (sqr(velpro_sd[i])-sqr(velpro_sd[i-2]) )/(velpro_s[i]-velpro_s[i-2]
velpro_sdd[VEL_PATH_NO-1] = velpro_sdd[VEL_PATH_NO-2];

/*plot the velocity profile in window 2*/
cell-id = winpar.wingid[4-1];
winset(cell_id);
linewidth(3);

color(BLACK);
clear();

/*draw the control points,velocity profile and acceleration profile*/
color(YELLOW);
for(i=1;i<=bzvel_.nconv;++i)circf(bzvel_.conv[i-1][0],bzvel_.conv[i-1][1],0.3);

color(RED);
move(velpro_s[0],velpro_sd[0],0.0);
for (i=2;i<=VEL_PATH_NO;++i) draw(velpro_s[i-1],velpro_sd[i-1],0.0);

color(BLUE);
move(velpro_s[0],velpro_sdd[0],0.0);
for (i=2;i<=VEL_PATH_NO;++i) draw(velpro_s[i-1],velpro_sdd[i-1],0.0);

swapbuffers();

storeglots = 1;
if(store_plots==1)
{
 /*fout = fopen("vel_s.mat","w");*/
 fout = fopen("vel_s.mat","w");
  for(i=1;i<=VEL_PATH_NO;++i) fprintf(fout,"%f\n",velpro_s[i-1]);
 fclose(fout);

 /*fout = fopen("vel_sd.mat","w");*/
 fout = fopen("vel_sd.mat","w");
  for(i=1;i<=VEL_PATH_NO;++i) fprintf(fout,"%f\n",velpro_sd[i-1]);
 fclose(fout);
```

```
}

}


/*************************************************************************/
/* generate_velocity_profile() using Bsplines for the acceleration profile */
/*************************************************************************/
generate_velocity_profile()
{
FILE *fout;
long  cell-id;
int  i,  ier;

int   storeglots;


/*the first and last control points for the x dir must be the same as for the
theta profile all others must lie in between these two*/
if (path-type==0) /*theta represented by bspline*/
{
  bzvel_.conv[0][0] = bz_.conp[0][0];
  bzvel_.conv[bzvel_.nconv-1][0] = bz_.conp[bz_.nconp-1][0];
}
else              /* (x(s),y(s)) represented by bspline*/
{
  bzvel_.conv[0][0] = s[0];
  bzvel_.conv[bzvel_.nconv-1][0] = s[PATH_NO-1];
`


/*if the x coordinate of any point is greater than that of the last point
 make thevalue that of the last point*/
for (i=2;i<=bzvel_.nconv-1;++i)
if (bzvel_.conv[i-1][0] >= bzvel_.conv[bzvel_.nconv-1] [0])
    bzvel_.conv[i-1][0] = bzvel_.conv[bzvel_.nconv-1][0];

/*if the x coordinate of any point is less than that of the first point
 make thevalue that of the first point*/
for (i=2;i<=bzvel_.nconv-1;++i)
if (bzvel_.conv[i-1][0] <= bzvel_.conv[0][0])
    bzvel_.conv[i-1][0] = bzvel_.conv[0][0];

/*generate the acceleration or velocity profile using bsplines*/
bspline_vel2_ (&VEL_PATH_NO,vel_s,vel_x,vel_xs,vel_xss,&ier_vel);
printf("vel_path_no = %d\n", VEL_PATH_NO);

/* compute the distance velpro_s[i-1] = vel_x[i-1][0]= maximumx coordinate*/
for (i=1;i<=VEL_PATH_NO;++i) velpro_s[i-1] = vel_x[i-1][0];

if (vel_type==0)
{
  /*compute sdd along the path*/
  for (i=1;i<=VEL_PATH_NO;++i) velpro_sdd[i-1]= vel_x[i-1][1];

  /*compute sd along the path by integrating the acceleration:
    sdd(s) = 0.5*d(sqr(sd))/ds = sqr(sd) = sqr(sd(0)) + int(2*sdd(s))     */

  velpro_sd[0] = sqr(constant_velocity_of_car);
  for(i=2;i<=VEL_PATH_NO;++i)
    velpro_sd[i-1] = velpro_sd[i-2] + 2*velpro_sdd[i-2]*(velpro_s[i-1]-velpro_s[i-2

  /*take squareroot*/
  for(i=1;i<=VEL_PATH_NO;++i)
    velpro_sd[i-1] = sqrt(velpro_sd[i-1]);
}
else /*vel_type==1 --> sd represented by a Bspline*/
```

```
{
  /*compute sd along the path*/
  for (i=1;i<=VEL_PATH_NO;++i) velpro_sd[i-1]= vel_x[i-1][1]

  /*compute sdd using finite differences*/
  for (i=1;i<=VEL_PATH_NO-1;++i)
  velpro_sdd[i-1]=
     0.5*(sqr(velpro_sd[i])-sqr(velpro_sd[i-1]))/(velpro_s[i]-velpro_s[i-1]);
  velpro_sdd[VEL_PATH_NO-1] = velpro_sdd[VEL_PATH_NO-2];
}

/*plot the velocity profile in window 2*/
cell-id = winpar.wingid[4-1];
winset(cell_id);
linewidth(3);

color(BLACK);
clear();

/*draw the control points,velocity profile and acceleration profile*/
color(YELLOW);
for(i=1;i<=bzvel_.nconv;++i)circf(bzvel_.conv[i-1][0],bzvel_.conv[i-1][1],0.3);

/*draw the point corresponding to the initial velocity*/
circf(velpro_s[0],constant_velocity_of_car,0.3);
/*draw the max constant velocity*/
color(CYAN);
move(velpro_s[0],constant_velocity_of_car,0.0);
draw(velpro_s[VEL_PATH_NO-1],constant_velocity_of_car,0.0);

color (RED);
move(velpro_s[0],velpro_sd[0],0.0);
for (i=2;i<=VEL_PATH_NO;++i) draw(velpro_s[i-1],velpro_sd[i-1],0.0);

color(BLUE);
move(velpro_s[0],velpro_sdd[0],0.0);
for (i=2;i<=VEL_PATH_NO;++i) draw(velpro_s[i-1],velpro_sdd[i-1],0.0);

/*store the previous profile (acceleration or velocity) */
linewidth(1);
color(GREEN);
move(vel_profile_at_switch[0][0],vel_profile_at_switch[0][1],0.0);
for (i=2;i<=VEL_PATH_NO;++i)
   draw(vel_profile_at_switch[i-1][0],vel_profile_at_switch[i-1][1],0.0);

/*draw the axes*/
color(RED);
move(0.0,0.0,0.0);
draw(120.0,0.0,0.0);
move(0.0,0.0,0.0);
draw(0.0,35.0,0.0);

swapbuffers();

storeglots = 1;
if(store_plots==1)
{
 fout = fopen("vel_s.mat","w");
  for(i=1;i<=VEL_PATH_NO;++i) fprintf(fout,"%f\n",velpro_s[i-1]);
 fclose(fout);

 fout = fopen("vel_sd.mat","w") ;
  for(i=1;i<=VEL_PATH_NO;++i) fprintf(fout,"%f\n",velpro_sd[i-1]);
 fclose(fout);
}
```

```
}


/*****************************************************************************/
/* generate_path_profile() generates theta, theta-s and theta-ss along the */
/* the path                                                                  */
/*****************************************************************************/
generate_path_profile(type_of_path)
int  type-ofgath;
{
int i;
long cell-id;
long ier;

int NS1, /*no of points along the first(before circ) straight line section*/
    NS2, /*no of points along the second(after circ) straight line section*/
    NC;  /*no of points along the circular arc*/

/*definitions for the turn section*/
float circ_centre[2], circ_radius;
float  ss_start[2], ss_finish[2], sf_start[2], sf_finish[2];
     /*initial and final points for the straight line sections*/
float anglegoint, angle-start, angle-finish, length-start, length-finish;
float sgn_theta_diff;


float anglex();   /*returns angle between -pi and +pi*/


if(type_of_path==0)
{
  /* generate path slopes */
  bspline2_(&PATH_NO,sangle,xangle,xangles,xangless,&ier_path) ;

  /* compute the distance s[i-1] = xangle[i-1][0]= maximumx coordinate*/
  for (i=1;i<=PATH_NO;++i) s[i-1] = xangle[i-1] [0];

  /*compute theta along the path*/
  for (i=1;i<=PATH_NO;++i) path.theta[i-1]= xangle[i-1] [1];

  /*compute theta-s using finite differences*/
  for (i=1;i<=PATH_NO-1;++i)
  path.theta_s[i-1]= (path.theta[i]-path.theta[i-1])/(s[i]-s[i-1]);
  path.theta_s[PATH_NO-1] = path.theta_s[PATH_NO-2];

  /*compute theta-ss using finite differences*/
  for (i=1;i<=PATH_NO-1;++i)
  path.theta_ss[i-1]= (path.theta_s[i]-path.theta_s[i-1])/(s[i]-s[i-1]);
  path.theta_ss[PATH_NO-1] = path.theta_ss[PATH_NO-2];


  /*generate the path in the work space*/
  generate_wspace_path();

  /*chekc path integration */
  check_path_integration();
}
else if(type_of_path==1)
{
  /*generate path in workspace*/
  bspline_w2_(&PATH_NO,s,x,xs,xss,&ier_path) ;

  /*compute theta(s) using the path*/
  for (i=1;i<=PATH_NO;++i) path.theta[i-1] = anglex(xs[i-1][0],xs[i-1][1]);

  /*compute theta-s and theta-ss using finite-differences*/
```

```
  for (i=1;i<=PATH_NO-1;++i)
  path.theta_s[i-1]= (path.theta[i]-path.theta[i-1])/(s[i]-s[i-1]);
  path.theta_s[PATH_NO-1] = path.theta_s[PATH_NO-2];

  /*smooth out theta-s 3- 15 -95*/
  /*
  for (i=2;i<=PATH_NO-2;++i)
  path.theta_s[i-1]= (path.theta_s[i-2]+path.theta_s[i])/2.0;

  for (i=2;i<=PATH_NO-2;++i)
  path.theta_s[i-1]= (path.theta_s[i-2]+path.theta_s[i])/2.0;
  */


  for (i=1;i<=PATH_NO-1;++i)
  path.theta_ss[i-1]= (path.theta_s[i]-path.theta_s [i-l])/(s[i]-s[i-1]);
  path.theta_ss[PATH_NO-1] = path.theta_ss[PATH_NO-2];

  /*smooth out thetass 3-35-95*/
  /*
  for (i=2;i<=PATH_NO-2;++i)
  path.theta_ss[i-1]= (path.theta_ss[i-2]+path.theta_ss[i])/2.0;
  */

  /*check path differentiation */
  check_path_integration();
}
else if(type_of_path==2)
{

  NS1 = 100;
  NC = 100;
  NS2 = 100;

  /*printf("at 1\n");*/
  /* define the center and radius of the circle*/
  circ_centre[0] =  50.0;   circ_centre[1] = 0.0;
  circ_radius = 10.0;

  /*define the start angle corresponding to the end of the first
                                                    straight line*/
  angle_start = PI/2.0;

  /*define the finish angle corresponding to the start of the second
                                                    straight line*/
  angle_finish = 0.0;

  /*define the length of the first straight line section*/
  length-start = 20.0;

  /*define the length of the second straight line section*/
  length-finish = 30.0;

  /* define the start and end points of section 1 - straight line*/
  ss_start[0] = circ_centre[0]+
     circ_radius*cos(angle_start) + length_start*cos(angle_start+PI/2.0);
  ss_start[1] = circ_centre[1]+
     circ_radius*sin(angle_start) + length_start*sin(angle_start+PI/2.0);

  ss_finish[0] = circ_centre[0]+circ_radius*cos(angle_start);
  ss_finish[1] = circ_centre[1]+circ_radius*sin(angle_start);

  /* define the start and end points of section 2 - straight line*/
  sf_start[0] = circ_centre[0]+circ_radius*cos(angle_finish);
  sf_start[1] = circ_centre[1]+circ_radius*sin(angle_finish);
```

```
sf_finish[0] = circ_centre[0]+
   circ_radius*cos(angle_finish) + length_finish*cos(angle_finish-PI/2.0);
sf_finish[1] = circ_centre[1]+
   circ_radius*sin(angle_finish) + length_finish*sin(angle_finish-PI/2.0);

/*define the number of points along the path*/
PATH-NO = NS1 + NC + NS2;

/*generate points, slopes etc for section 1*/
for(i=1;i<=NS1;++i)
{
  x[i-1][0] = (SS_finish[0]*(i-1) + ss_start[0]*(NS1-i))/(NS1-1);
  x[i-1][1] = (ss_finish[1]*(i-1) + ss_start[1]*(NS1-i))/(NS1-1);

  /*printf("S1, X=   %f, %f\n", x[i-1][0], x[i-1][1]);*/


  xs[i-1][0] = (ss_finish[0] -ss_start[0])/length_start;
  xs[i-1][1] = (ss_finish[1]-ss_start[1])/length_start;

  /*printf("S1, XS=   %f, %f\n", xs[i-1][0], xs[i-1][1]);*/

  xss[i-1][0] = 0.0;
  xss[i-1][1] = 0.0;

  s[i-1]     = length-start*(i-l)/(NSl-1);
}

/*printf("at 2\n");*/
/*compute theta(s) for section 1*/
for (i=1;i<=NS1;++i) path.theta[i-1] = angle_start-PI/2.0; /*anglex(xs[i-1][0],xs

/*compute theta-s and theta-ss using finite-differences*/
for (i=1;i<=NS1-1;++i)
path.theta_s[i-1]= (path.theta[i]-path.theta[i-1])/(s[i]-s[i-1]);
path.theta_s[NS1-1] = path .theta_s[NS1-2];

for (i=1;i<=NS1-1;++i)
path.theta_ss[i-1]= (path.theta_s[i]-path.theta_s[i-1])/(s[i]-s[i-1]);
path.theta_ss[NS1-1] = path.theta_ss[NS1-2];

/*generate points, slopes etc for circular section*/
for(i=1;i<=NC;++i)
{
  anglegoint  = (angle-finish*(i-1)  +  angle-start*(NC-i))/(NC-1);
  if( (anglegoint-angle-start) >= 0.0) sgn_theta_diff = 1.0;
  else                                 sgn_theta_diff = -1.0;

  x[NS1+ i-l] [0] = circ_centre[0]+circ_radius*cos(angle_point);
  x[NS1+ i-l] [1] = circ_centre[1]+circ_radius*sin(angle_point);

  xs[NS1+ i-l] [0] = -sin(angle_point)/sgn_theta_diff;
  xs[NS1+ i-1][1] = cos(angle_point)/sgn_theta_diff;

  xss[NS1+ i-l] [0] = 0.0;
  xss[NS1+ i-1][1] = 0.0;

  s[NS1+ i-l]      = s[NS1-1]+circ_radius*abs(angle_point-angle_start);

  path.theta[NS1+ i-l] = anglegoint - PI/2.0; /*anglex(xs[NS1+ i-1][0],xs[NS1+ i
}

/*compute theta(s) for section 1*/
/* see above.
for (i=1;i<=NC;++i)
    path.theta[NS1+ i-l] = anglex(xs[NS1+ i-l][0],xs[NS1+ i-l][1]);
```

```
      */

    /*compute theta-s and theta-ss using finite-differences*/
    for (i=1;i<=NC-1;++i)
    path.theta_s[NS1+ i-1]= (path.theta[NS1+ i]-path.theta[NS1+ i-1 ])/(s[NS1+ i]-s[NS
    path.theta_s[NS1+ NC-1] = path.theta_s[NS1+ NC-21;

    for (i=1;i<=NC-1;++i)
    path.theta_ss[NS1+ i-1]= (path.theta_s[NS1+ i]-path.theta_s[NS1+ i-1])/(s[NS1+ i]
    path.theta_ss[NS1+ NC-1] = path.theta_ss[NS1+ NC-2];


    /*generate points, slopes etc for section 2*/
    for(i=1;i<=NS2;++i)
    {
      x[NS1+NC+ i-1][0] = (sf_finish[0]*(i-1) + sf_start[0]*(NS2-i))/(NS2-1);
      x[NS1+NC+ i-1][1] = (sf_finish[1]*(i-1) + sf_start[1]*(NS2-i))/(NS2-1);

      xs[NS1+NC+ i-1][0] = (sf_finish[0]-sf_start[0])/length_finish;
      xs[NS1+NC+ i-1] [1] = (sf_finish[1]-sf_start[1])/length_finish;

      xss[NS1+NC+ i-1][0] = 0.0;
      xss[NS1+NC+ i-1][1] = 0.0;

      s[NS1+NC+ i-1]      = s[NS1+NC-1]+ length_finish*(i-1)/(NS2-1);


    /*compute theta(s) for section 1*/
    for (i=1;i<=NS2;++i)
        path.theta[NS1+NC+ i-1] = angle_finish-PI/2.0; /*anglex(xs[NS1+NC+ i-1][0],xs

    /*compute theta-s and theta-ss using finite-differences*/
    for (i=1;i<=NS2-1;++i)
    path.theta_s[NS1+NC+ i-1]= (path.theta[NS1+NC+ i]-path.theta[NS1+NC+ i-1])/(s[NS1
    path.theta_s[NS1+NC+NS2  -1] = path.theta_s[NS1+ NC +NS2 -21;

    for (i=1;i<=NS2-1;++i)
    path.theta_ss[NS1+NC+ i-1]= (path.theta_s[NS1+NC+ i]-path.theta_s[NS1+NC+ i-11)/(:
    path.theta_ss[NS1+ NC+NS2 -1] = path.theta_ss[NS1+ NC+NS2 -21;



    /*check path*/
    check_path_integration();

}
else
{
}

/*print check*/
/*
for (i=1;i<=PATH_NO;++i)
printf("theta = %f\n",path.theta[i-1]);
getchar();
for (i=1;i<=20;++i)
printf(" \n");
getchar();
for (i=1;i<=PATH_NO;++i)
printf("theta_s = %f\n",path.theta_s[i-1]);
getchar();
for (i=1;i<=20;++i)
printf (" \n");
getchar();
for (i=1;i<=PATH_NO;++i)
printf("theta_ss = %f\n",path.theta_ss[i-1]);
```

```
getchar();
*/

cell_id = winpar.wingid[3-1]  ;
winset(cell_id);
color(BLACK);
clear();

/*plot theta in window 3*/
color(YELLOW);
plot_(s,path.theta,PATH_NO,3);

color(WHITE);
plot_(s,theta_actual,PATH_NO,3);

/*plot theta-s in window 3*/
color(BLUE);
plot_(s,path.theta_s,PATH_NO,3);

/*plot theta-ss in window 3*/
color(MAGENTA);
plot_(s,path.theta_ss,PATH_NO,3);


swapbuffers();
}

/*****************************************************************************/
/*              generate path inworkspace by integrating theta               */
/*****************************************************************************/
generate_wspace_path()
{

int i;

/*initial point*/
x[0][0] = init_point[0];
x[0][1] = init_point[1];

/*integrate using theta: x[i-1][0] = init_point[0] + int_(0,s) cos(theta)*/
for(i=2;i<=PATH_NO;++i)
{
  /* xcoordinates */
  x[i-1][0] = x[i-2][0] + cos(path.theta[i-2])*(s[i-1]-s[i-2]);

  /* y coordinates */
  x[i-1][1] = x[i-2][1] + sin(path.theta[i-2])*(s[i-1]-s[i-2]);
}


}
generate_wspace_path_2() /*uses runge kutta 3rd order integration*/
{

 int i;

 /*initial point*/
 x[0][0] = init_point[0];
 x[0][1] = init_point[1];

 /*integrate using theta: x[i-1][0] = init_point[0] + int_(0,s) cos(theta)*/
 for(i=2;i<=PATH_NO;++i)
 {
   /* xcoordinates */
   x[i-1][0] = x[i-2][0] + cos(path.theta[i-2])*(s[i-1]-s[i-2]);
```

```
   /* y coordinates */
   x[i-1][1] = x[i-2][1] + sin(path.theta[i-2])*(s[i-1]-s[i-2]);
 }


}


/*************************************************************************/
/*   program to check if the path integration is accurate                */
/*************************************************************************/

check_path_integration()
{
  int i;
  float anglex();
  /* compute theta(s) by differentiating the path to find slope */
    for (i=1;i<=PATH_NO-1;++i)
    {
      xs[i-1][0] = (x[i][0]-x[i-1][0])/(s[i]-s[i-1]);
      xs[i-1][1] = (x[i][1]-x[i-1][1])/(s[i]-s[i-1]);
    }
    xs[PATH_NO-1][0] = xs[PATH_NO-2][0];
    xs[PATH_NO-1][1] = xs[PATH_NO-2][1];

    for (i=1;i<=PATH_NO;++i) theta_actual[i-1] = anglex(xs[i-1][0],xs[i-1][1]);
}


/*************************************************************************/
/*      computes the state dynamics for alpha integration                */
/*************************************************************************/
/*
stateq(t,x,xprime)
float t,x[2],xprime[2];
{
xprime[0] = x[1];
xprime[1] = - x[0];
}
*/

/*to check integration of the path*/
stateq(dist,x,xprime)
float dist,x[2],xprime[2];
{
int i,  s_index;
float theta_at_s;

for (i=1;i<=PATH_NO-1;++i)
{
  if ( (dist>=s[i-1]) && (dist<s[i]) ) s-index = i;
}

/*find theta at s by linear interpolation*/
theta-at-s =
path.theta[s_index-1] +
(dist-s[s_index-1])*(path.theta[s_index]-path.theta[s_index-1])/(s[s_index]-s[s_ind


/*compute the derivative*/
xprime[0] = cos(theta_at_s);
xprime[l] = sin(theta_at_s); /* sin(t);*/
}

/*for the integration of alpha with respect to s*/
state-alpha(dist,x,xprime)
```

```
float dist,x[2],xprime[2];
{
float get_quantity_at_s();
float f_alpha_dd();

float theta-at-s, thetas-at-s, thetass_at_s;
float sd-at-s, sdd-at-s, thetad_at_s, thetadd_at_s;

float Crear; /*force constant on rear wheel*/

Crear = 80000; /*350.0;*/ /*400.0;*/


/*find all required quantities at s = dist*/
sd_at_s      = get_quantity_at_s("sdv",dist);
sdd at_s     = get_quantity_at_s("sdd",dist);
theta at_s   = get_quantity_at_s("the",dist);
thetas_at_s  = get_quantity_at_s("ths",dist);
thetass_at_s = get_quantity_at_s("tss",dist);

thetad_at_s   = sd_at_s*thetas_at_s;
thetadd-at-s  = sdd_at_s*thetas_at_s + sqr(sd_at_s)*thetass_at_s;


/*compute the derivative*/
xprime[0] = x[1];
xprime[1] = (f_alpha_dd(dist, sd-at-s, sdd-at-s, x[0], x[1]*sd_at_s, thetad_at_s, tl
}

state-alpha-kin(dist,y,yprime)
float dist,y[2],yprime[2];
{
float get-quantity-at-so;

float thetas-at-s;

thetas-at-s = get_quantity_at_s("ths",dist);

yprime[0] = - sin(y[0])/car.d - thetas-at-s;
yprime[1] = 0.0;


I


/*************************************************************************/
/*      generate_alpha() generates alpha by forward difference integration   */
/*************************************************************************/
generate_alpha()
{
 /*first try out case with sd = constant -> sdd = 0*/
 float path_alpha[500], path_alpha_s[500], path_alpha_ss[500];
 float thetadd, thetad;
 float alphad;
 float sd;
 int  i;

 float f_alpha_dd();

 sd = 1;

 /*initial conditions on alpha and alphas*/
 path_alpha[0]   = 0.0;
 path_alpha_s[0] = 0.0;

 /*compute alpha-ss along the path*/
```

```
 for (i=1;i<=PATH_NO;++i)
 {
 alphad = path-alpha-s[i-ll*sd;
 thetad = path.theta_s[i-1]*sd;
 thetadd = path.theta_ss[i-1]*sqr(sd);

 path-alpha-ss[i-l] = f_alpha_dd(s[i-1], sd, 0.0, path_alpha[i-1], alphad, thetad,

 path-alpha_s[i] = path-alpha_s[i-1] + (s[i]-s[i-1])*path_alpha_ss[i-1];
 path_alpha[i]   = path_alpha[i-1]   + (s[i]-s[i-1])*path_alpha_s[i-1];
 }
}



float get-quantity-at-s(qty,dist)
char qty[3];
float dist;
{
int i, s index;
float quantity;
float get_vel_acc_at_s();

int point-flag;

point-flag = 0;
/*find the indices of the points encompassing the given distance*/
for (i=1;i<=PATH_NO-1;++i)
{
  if ( (dist>=s[i-1]) && (dist<s[i]) ) {s-index = i; point_flag=1;}
}

if(point_flag==1)
{
   if ( (qty[0]=='t') && (qty[1]=='h') && (qty[2]=='e') )
   {
     /*if qty = "the" return theta*/
     quantity = path.theta[s_index-1] +
     (dist-s[s_index-1])*(path.theta[s_index]-path.theta[s-index-l])/(s[s_index]-s[
     return(quantity);
   }
   else if ( (qty[0]=='t') && (qty[1]=='h') && (qty[2]=='s') )
   {
     /*if qty = "ths" return theta-s*/
     quantity = path.theta_s[s index-l] +
     (dist-s[s_index-1])*(path.theta_s[s_index]-path.theta_s[s_index-1])/(s[s_index
     return(quantity);
   }
   else if ( (qty[0]=='t') && (qty[1]=='s') && (qty[2]=='s') )
   {
     /*if qty = "tss" return theta_ss */
     quantity = path.theta_ss[s_index-1] +
     (dist-s[s_index-1])*(path.theta_ss[s_index]-path.theta_ss[s_index-1])/(s[s_ind
     return(quantity);
   }
   else if ( (qty[0]=='s') && (qty[1]=='d') && (qty[2]=='v') )
   {
     /* if qty = "sdv" return sd*/
     /*quantity = constant-velocity-of-car: */
     quantity = get_vel_acc_at_s("vel",dist);
     return(quantity);
   }
   else if ( (qty[0]=='s')  && (qty[1]=='d') && (qty[2]=='d') )
   {
     /*if qty = "sdd" return sdd*/
```

```
      /*quantity = 0.0; */
      quantity = get_vel_acc_at_s("acc",dist);
      return(quantity);
    }
    else
    {
      printf("quantity unrecognizable!! ERROR !\n");
      getchar();
      return(-1.0);
    }
}
else /*return all quantities at the end of the path*/
{
    if ( (qty[0]=='t') && (qty[1]=='h') && (qty[2]=='e') )
    {
      quantity = path.theta[PATH_NO-1];
      return(quantity);
    }
    else if ( (qty[0]=='t') && (qty[1]=='h') && (qty[2]=='s') )
    {
      quantity = path.theta_s[PATH_NO-1];
      return(quantity);
    }
    else if ( (qty[0]=='t') && (qty[1]=='s') && (qty[2]=='s') )
    {
      quantity = path.theta_ss[PATH_NO-1];
      return(quantity);
    }
    else if ( (qty[0]=='s') && (qty[1]=='d') && (qty[2]=='v') )
    {
      quantity = get_vel_acc_at_s("vel",s[PATH_NO-1]);
      return(quantity);
    }
    else if ( (qty[0]=='s') && (qty[1]=='d') && (qty[2]=='d') )
    {
      quantity = get_vel_acc_at_s("acc",s[PATH_NO-1]);
      return(quantity);
    }
    else
    {
      printf("quantity unrecognizable!!! ERROR !\n");
      getchar();
      return(-1.0);
    }
}

}



/*************************************************************************/
/*    returns the velocity and acceleration at a given dist usng the     */
/*    bspline representing the velocity profile                          */
/*************************************************************************/
float get_vel_acc_at_s(qty,dist)
char  qty[3];
float dist;
{
int i,  s index;
float  quantity;

int  point-flag;

point-flag = 0;

/*find the indices of the points encompassing the given distance*/
```

```
for (i=1;i<=VEL_PATH_NO-1;++i)
{
  if( (dist>=velpro_s[i-1]) && (dist<velpro_s[i]) ) {s-index = i; point_flag=1;}
}

if (point_flag==1)
{
   if  ( (qty[0]=='v') && (qty[1]=='e') && (qty[2]=='l') )
   {
     /*if qty = "vel" return sd*/
     quantity = velpro sd[s_index-1] +
     (dist-velpro_s[s_index-1])*(velpro_sd[s_index]-velpro_sd[s_index-1])/(velpro_s
     return(quantity);
   }
   else  if  ( (qty[0]=='a') && (qty[1]=='c') && (qty[2]=='c') )
   {
     /*if qty = "acc" return sdd*/
     quantity = velpro_sdd[s_index-1] +
     (dist-velpro_s[s_index-1])*(velpro_sdd[s_index]-velpro_sdd[s_index-1])/(velpro
     return(quantity);
   }
   else
   {
     printf ("quantity unrecognizable!! ERROR !\n");
     getchar();
     return(-1.0);
   }
}
else /*return quantities at the end of the velocity profile*/
{
   if  ( (qty[0]=='v') && (qty[1]=='e') && (qty[2]=='l') )
   {
     quantity = velpro_sd[VEL_PATH_NO-1];
     return(quantity);
   }
   else  if  ( (qty[0]=='a') && (qty[1]=='c') && (qty[2]=='c') )
   {
     quantity = velpro_sdd[VEL_PATH_NO-1];
     return(quantity);
   }
   else
   {
     printf("quantity unrecognizable!! ERROR !\n");
     getchar();
     return(-1.0);
   }
}

}
```

```
/*****************************************************************************/
/* dugoff.c: program to compute the forces given by the dugoff model         */
/*****************************************************************************/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define sqr(x) ((x)*(x))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define PI             3.141592654

/*type of model used*/
extern int model-flag; /*0 --> linear model
                         1 --> dugoff model with exhaustive computation
                               of ellipse
                         2 --> dugoff model with simplified computation
                               of ellipse
                        */

/*dugoff model ellipse major and minor axes values for plotting?*/
float dugoff_ellipse_Fxmax,
      dugoff_ellipse_Fymax;

/*finds the best fit ellipse satisfying the dugoff tire force model*/
find-ellipse(alpha, Fxmax, Fymax)
float alpha: /*slip angle of the rear tire*/
float *Fxmax, *Fymax;
{
FILE *fout;

int  i;
int   loop-flag;
int   npoints;

float s[100];             /*slip ratios*/
float Fx[300], Fy[300];   /*forces*/
float flamda();

/*tire parameters*/
float W, mu0, Cal, Cs, eps, V;

/*dugoff model paramters*/
float num, den, fl, lam;

/*sakai model parameters*/
float Kx, Fz, mus, mud, q, h;

float slip;

/*define the tire parameters*/
w   = 1550*10/2.0; /*3875;*/
mu0 = 0.85;
Cal = 80000.0;  /*133000;*/
CS  = 74000.0; /*186000;*/
eps = 0.0031;
V   = 20.0;

/* compute corresponding Fx and Fy */
/*
printf("alpha in degrees = ?\n");
scanf("%f",&alpha);
printf("alpha (deg) =  %f\n",alpha);
getchar();
alpha = alpha*PI/180.0;
*/
```

```
if (model_flag==1)
{
   /*search for the maximum forces over the range of slips*/
   *Fxmax = 0.0;
   *Fymax = 0.0;

   i= 0;
   for (slip= -0.99; slip<-0.99; slip = slip+0.01)
   {
      i = i+1;

      num = mu0*W*( 1-eps*V*sqrt(sqr(slip)+sqr(tan(alpha))) )*(l-slip);
      den = 2*sqrt( sqr(Cs*slip) + sqr(Cal*tan(alpha)) );
      lam = numlden:

      fl = flamda(lam);

      Fx[i-1] = Cs*slip*fl/(1-slip);
      Fy[i-1] = Cal*tan(alpha)*fl/(1-slip);

      if(abs(Fx[i-1])>*Fxmax) *Fxmax = abs(Fx[i-1]);
      if(abs(Fy[i-1])>*Fymax) *Fymax = abs(Fy[i-1]);

      /*
      printf("Fx(%d) = %f, Fy(%d) = %f\n", i, Fx[i-1], i, Fy[i-1]);
      */
   }
   npoints = i;

   /*compute the max value of Fx*/
   /*
   num = mu0*W*( 1-eps*V*sqrt(1.0+sqr(tan(alpha))) );
   den = 2*sqrt( sqr(Cs) + sqr(Cal*tan(alpha)) );
   *Fxmax = 2.0*Cs*num/den;
   */
   /*
   printf ("Fxmax= %f\n",*Fxmax);
   printf("Fymax= %f\n",*Fymax);
   */

   /*store the forces for plotting*/
   /*
   fout = fopen ("Fx.dat","w") ;
   for (i=1;i<=npoints;++i) fprintf(fout,"%f\n",Fx[i-1]);
   fclose(fout);
   */

   /*
   fout = fopen("Fy.dat","w");
   for (i=1;i<=npoints;++i) fprintf(fout,"%f\n",Fy[i-1]);
   fclose(fout);
   */

   /*store ellipse major and minor axes for plotting*/
   dugoff_ellipse_Fxmax = *Fxmax;
   dugoff_ellipse_Fymax = *Fymax;
}
else if(model_flag==2)
{
   /*compute Fymax, which occurs at slip = 0.0*/
   slip = 0.0;
   num = mu0*W*( 1-eps*V*sqrt(sqr(slip) +sqr(tan(alpha))) )*(l-slip);
   den = 2*sqrt( sqr(Cs*slip) + sqr(Cal*tan(alpha)) );
   lam = num/den;
   fl = flamda(lam);
```

```
    *Fymax = abs(Cal*tan(alpha) ● fl/(l-slip));

    /*compute Fxmax*/
    num = muO*W*( 1-eps*V*sqrt(1.0+sqr(tan(alpha))) );
    den = 2*sqrt( sqr(Cs) + sqr(Cal*tan(alpha)) );
    *Fxmax = 2.0*Cs*num/den;

    /*
    printf("Fxmax= %f\n",*Fxmax);
    printf("Fymax= %f\n",*Fymax);
    */

    /*store ellipse major and minor axes for plotting*/
    dugoff_ellipse_Fxmax = *Fxmax;
    dugoff_ellipse_Fymax = *Fymax;
}
else
{
}

}



/* computes flambda for the dugoff model*/
float flamda (lam)
float lam:
{
 float fl;

 if (lam < 1)
    fl = lam*(2-lam);
 else if(lam >= 1)
    {fl = 1;  /*printf ("flamda saturated\n");*/}
 else
    {}

 return(fl);
}







/*****************************************************************************/
/*      dugoff_force() : returns the force given by the dugoff model         */
/*****************************************************************************/
float dugoff_force(Fx, alpha-rear)
float Fx;          /*tractive force*/
float  alpha-rear; /* rear tire slip angle */
{
float  Fxmax, Fymax;
float f_fn();
float Crear;
float  f-rear-actual;

find-ellipse(alpha-rear,  &Fxmax, &Fymax);

if( abs(Fx) > abs(Fxmax) )
{
  printf("error in dugoff force!!, Fx = %f, Fxmax = %f\n", Fx,  Fxmax);
  getchar();
I

Crear = 80000.0;
```

```
/*use the same sign as the actual tire force model*/

f-rear-actual = f_fn(alpha_rear,Crear);

/*compute the force*/
if (f-rear-actual >=0.0)
   return(  Fymax*sqrt( 1 -  sqr(Fx/Fxmax)  )  );
else
   return(  -Fymax*sqrt( 1 -  sqr(Fx/Fxmax)  )  );

}
```

```
/*****************************************************************************
written by Satish
drw_path_w draws the path in the workspace
*****************************************************************************/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "runge.h"

#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define PI              3.141592654

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                  ) winpar ;
extern float s[500],x[500][2],xs[500][2],xss[500][2];
extern struct t {int nconp;  float conp[20][2];} bz_;
extern struct tw {int nconp;  float conp[20][2];} bzw_;
extern int PATH-NO;
extern struct vehicle {float mass, Ic, d, alpha, alpha-f, alpha_r, Ft, Beta,
                       fnr,  fnf, width,  wheel-length, wheel_width;} car;
extern int curr_pos_of_car; /*index of current car position*/

extern struct paths {float theta[500],theta_s[500],theta_ss[500];} path;

/*to represent theta by a bspline */
extern float sangle[500],xangle[500][2],xangles[500][2],xangless[500][2];
extern float init_point[2];

/*actual value of theta to check accuracy of path integration */
extern float theta_actual[500];

/*alpha*/
struct paths2 {float alpha[500],alpha_s[500],alpha_ss[500];} pdyn;
/*steering angle and driving force*/
extern struct forces {float beta[1000], drive[1000], dis[1000]; int np;} force;


/*to switch between path represetnations*/
extern int path-type;

/*store theta when switching path type*/
extern float pos_profile_at_switch[400][2];
extern int   pos_profile_no;

/*---dugoff model ellipse major and minor axes values for plotting?---*/
float dugoff_ellipse_Fxmax,
      dugoff_ellipse_Fymax;
/*-----------------------------------------------------------------*/

drw_path_w(cell_no)
int cell_no;
{
 long  cell-id; /*cell id*/
 int   ';
 float theta, theta-s, theta-ss;
 float anglex();
 float get-control-at_s();
 char txt[40]; /*to display numbers*/
```

```
 /*---for dugoff model---*/
 float f_alpha_r();
 float rear force dugoff; /*rear tire force by dugoff model*/
 float rear_force_sat;    /*rear tire force by saturation model*/
 float dugoff_force();
 float f fn();
 float Gear;
 float alpha-rear:

 void  ellipsef();
 /*---------------------*/

/*calculate the angle theta = slope of the path at the current point*/
 theta = path.theta[curr_pos_of_car-1];
 theta-s  = path.theta_s[curr_pos_of_car-1];
 theta-ss = path.theta_ss[curr_pos_of_car-1];
 /*printf("theta = %f\n", theta); */

 car.alpha = pdyn.alpha[curr_pos_of_car-1];
 car.Beta =  get_control_at_s("str",s[curr_pos_of_car-1]);
             /*force.beta[curr pos of-car-l];*/
 car.Ft   =  get-control_at_s("drv",s[curr_pos_of_car-1]);
             /*force.drive[curr_pos_of_car-1];*/

/*-------------------------------------------------------------------------*/
/*-------compute the ellipse representing the dugoff model----------------*/
/*-------at  the current point for plotting--------------------------------"/

  ,'*compute alpha-rear*/
  alpha_rear = f_alpha_r(s[curr_pos_of_car-1], 1.0, car.alpha, 0.0, pdyn.alpha_s[cu

  /*compute dugoff force and store major axes in dugoff_ellipse_Fxmax etc.*/
  /*
  rear-force-dugoff  = dugoff_force(car.Ft,alpha_rear);
  printf("ellipse axes = %f, %f\n", dugoff_ellipse_Fxmax, dugoff_ellipse_Fymax);
  */

  /*compute sat. model force*/
  Crear = 80000.0;
  rear-force-sat = f_fn(alpha_rear,Crear);


/*-------------------------------------------------------------------------*/

 cell-id = winpar.wingid[cell_no-1];
 winset(cell_id);
 linewidth(3);

 color(BLACK);
 clear();

 /*draw the workspace axis*/
 color(RED);
 move(0.0,0.0,0.0);
 /*draw(35.0,0.0,0.0);*/
 draw(135.0,0.0,0.0);
 move(0.0, -15.0,0.0);
 draw(0.0,15.0,0.0);

 /* draw the lines representing points where lane changes begins and ends*/
 color(BLUE);
 /*
 move(lO.0, -15.0,0.0);
 draw(10.0,15.0,0.0);
 move(20.0, -15.0,0.0);
 draw(20.0,15.0,0.0);
```

```
move(15.0,-15.0,0.0);
draw(15.0,15.0,0.0);
*/

move(50.0,-15.0,0.0);
draw(50.0,15.0,0.0);
move(70.0,-15.0,0.0);
draw(70.0,15.0,0.0);
move(90.0,-15.0,0.0);
draw(90.0,15.0,0.0);

if(path_type==0) /*theta(s) represented by a bspline*/
{
  /*show the initial point in the workspace*/
  color(YELLOW);
  circf(init_point[0],init_point[1],0.3);
}
else            /*(x(s),y(s)) represented by a bspline*/
{
  /*draw the control points in the workspace*/
  color(YELLOW);
  for (i=1;i<=bzw_.nconp;++i) circf(bzw_.conp[i-1][0],bzw_.conp[i-1][1],0.2);
}

/*draw the path obtained by integrating theta*/
color(RED);
move(x[0][0],x[0][1],0.0);
for (i=2;i<=PATH_NO;++i) draw(x[i-1][0],x[i-1][1],0.0);


/* draw the car at its current position*/
pushmatrix();
  translate(x[curr_pos_of_car-1][0], x[curr_pos_of_car-1][1], 0.0);
  rot(180*(theta+car.alpha)/PI, 'z');
  pushmatrix();
    drw_body();
  popmatrix();
popmatrix();


/*-------------------------------------------------------------------------*/
/*draw the ellipse corresponding to the dugoff model*/
/*
color(CYAN);
ellipsef(50.0,  5.0, dugoff_ellipse_Fxmax/200, dugoff_ellipse_Fymax/800);
color(YELLOW);
circf(50.0,  5.0+rear_force_dugoff/800,0.2);
circf(50.0+car.Ft/200, 5.0,0.2);
circf(50.0+car.Ft/200, 5.0+rear_force_dugoff/800,0.2);
color(GREEN);
circf(50.0,  5.0+rear_force_sat/800,0.2);
*/
/*-------------------------------------------------------------------------*/

/* display numbers in the window*/
 color (WHITE);
/*theta-ss*/
 real_char(txt,7,theta_ss);
 cmov(winpar.wincoord[0][cell_no-1] + 2.05,winpar.wincoord[2][cell_no-1] + 7*0.43,0
 charstr("   theta-ss  =  ");
 charstr(txt);

/*theta_s*/
 real_char(txt,7,theta_s);
 cmov(winpar.wincoord[0][cell_no-1] + 2.05,winpar.wincoord[2][cell_no-1] + 6*0.43,0
 charstr("   theta_ s  =  ");
```

```
 charstr(txt);

/*theta*/
 real_char(txt,7,180*theta/PI);
 cmov(winpar.wincoord[0][cell_no-1] + 2.05,winpar.wincoord[2][cell_no-1] + 5*0.43,0
 charstr("   theta    = ");
 charstr(txt);

/*alpha*/
 real_char(txt,7,180*car.alpha/PI);
 cmov(winpar.wincoord[0][cell_no-l] + 2.05,winpar.wincoord[2][cell_no-1] + 4*0.43,0
 charstr("   alpha    = ");
 charstr(txt);

/*Beta*/
 real_char(txt,7,180*car.Beta/PI);
 cmov(winpar.wincoord[0][cell_ no-l] + 2.05,winpar.wincoord[2][cell_no-1] + 3*0.43,0
 charstr("   Beta     = ");
 charstr(txt);

/*Ft*/
 real_char(txt,7,car.Ft);
 cmov(winpar.wincoord[0][cell_no-1] + 40.0,winpar.wincoord[2][cell_no-1] + 3*0.43,0
 charstr("   Ft   =   ");
 charstr(txt) ;

/*alphaf*/
 real_char(txt,7,180*car.alpha_f/PI);
 cmov(winpar.wincoord[0][cell_no-l] + 2.05,winpar.wincoord[2][cell_no-1] + 2*0.43,0
 charstr("   alphaf   =   ");
 charstr(txt);

/*alphar*/
 real_char(txt,7,180*car.alpha_r/PI);
 cmov(winpar.wincoord[0][cell_no-1] + 2.05,winpar.wincoord[2][cell_no-1] + 0.43,0.0
 charstr("   alphar   =   ");
 charstr(txt);


 /*plot path cmputed using runge-kutta integration*/
 color(MAGENTA);
 move(yout[0][0], yout[0][1], 0.0);
 for (i=2;i<=npoints;++i) draw(yout[i-l][0], yout[i-1][1], 0.0);

 swapbuffers();

}

/*****************************************************************************/
/*         plots the bspline representing theta                              */
/*****************************************************************************/

drw_theta(cell_no)
int cell_no:
{
 long  cell-id; /*cell id*/
 int   i;
 char txt[40]; /*to display numbers*/


 cell-id = winpar.wingid[cell_no-1];
 winset(cell_id);
 linewidth(3);

 color(BLACK);
 clear();
```

```
 if (path-type==0) /*theta(s) represented using bspline*/
 {
    /*draw the control points*/
    color(YELLOW);
    for (i=1;i<=bz_.nconp;++i) circf(bz_.conp[i-1][0],bz_.conp[i-l][1],0.05);
 }

 /*draw the path slopes */
 color (RED);
 move(s[0],path.theta[0],0.0);
 for (i=2;i<=PATH_NO;++i) draw(s[i-1],path.theta[i-1],0.0);

 /*draw the actual value of theta(s) obtained by differentiating the path*/
 color(WHITE);
 move(s[0],theta_actual[0],0.0);
 for (i=2;i<=PATH_NO;++i) draw(s[i-1],theta_actual[i-1],0.0);

 /*store the previous theta profile*/
 linewidth(1);
 color(GREEN);
 move(pos_profile_at_switch[0][0],pos_profile_at_switch[0][1],0.0);
 for (i=2;i<=pos_profile_no;++i)
    draw(pos_profile_at_switch[i-1][0],pos_profile_at_switch[i-1][1],0.0);


 color(RED);
 /*draw the axes to indicate limits on theta and s*/
 linewidth(1);
 move( 0.0,-1.0*PI,0.0);
 draw(150.0, -1.0*PI,0.0);
 move( 0.0,1.0*PI,0.0);
 draw(150.0,1.0*PI,0.0);
 move(O.O, -1.0*PI,0.0);
 draw(0.0, 1.0*PI,0.0);
 move (150.0, -1.0*PI,0.0);
 draw(150.0, 1.0*PI,0.0);

 /*draw lines indicating PI/2 and -PI/2*/
 move ( 0.0,-0.5*PI,0.0);
 draw(150.0, -0.5*PI,0.0);
 move( 0.0,0.5*PI,0.0);
 draw(150.0,0.5*PI,0.0);

 /*draw the 0 degree line*/
 color(BLUE);
 move( 0.0,-0.0,0.0);
 draw(150.0,-0.0,0.0);


 swapbuffers();

}

/* draws the body of the car at (0.0,0.0) */
drw_body()
{
/*draw the frame*/
color(BLUE) ;
rectf(-car.d, -car.width, car.d, car.width);

/*draw the body frame axis in white*/
color (WHITE);
move(0.0,0.0,0.0);
draw(0.5*car.d,0.0,0.0);
move(0.0,0.0,0.0);
```

```
draw(0.0,0.5*car.d,0.0);

/*draw the front wheel*/
pushmatrix();
  translate(car.d,0.0,0.0);
  pushmatrix();
    rot(180*car.Beta/PI,'z');
    pushmatrix();
      color(YELLOW);
      rectf(-0.5*car.wheel_length, -car.wheel_width, 0.5*car.wheel_length,
            car.wheel_width);
      color(BLUE);/*draw the wheel axis in blue*/
      move(0.0,0.0,0.0);
      draw(0.5*car.wheel_length,0.0,0.0);
      move(0.0,0.0,0.0);
      draw(0.0,1.5*car.wheel_width,0.0);
    popmatrix();
  popmatrix();
popmatrix();

/*draw the back wheel*/
color (MAGENTA);
pushmatrix();
  translate(-car.d,0.0,0.0);
  rectf(-0.5*car.wheel_length, -car.wheel_width, 0.5*car.wheel_length,
        car.wheel_width);
popmatrix();
}


/*************************************************************************/
/*                        plotting programs                              */
/*************************************************************************/
plot_(x,y,no,cell_no) /*plots y(x) */
float *x,*y;
int no,cell_no:
{
float lxmin, lxmax, lymin, lymax;  /*lengths of the box edges*/
float xmax,ymax,max_array();
float xmin,ymin,min_array();
float sc_x,sc_y;/*scaling factor*/
long cell-id;
int i;

lxmin = winpar.wincoord[0][cell_no-1];
lxmax = winpar.wincoord[1][cell_no-1];
lymin = winpar.wincoord[2][cell_no-1];
lymax = winpar.wincoord[3][cell_no-1];

xmax = max_array(x,no);
yma x = max_array(y,no);
xmin = min_array(x,no);
ymin = min_array(y,no);

sc_x = (lxmax-lxmin)/(xmax-xmin);
sc_y = (lymax-lymin)/(ymax-ymin);

/*
cell-id = winpar.wingid[cell_no-1];
winset(cell_id);
color(BLACK);
clear();
*/

/*draw the box*/
/*
```

```
color(RED);
move2(lxmin-0.1,0.0);
draw2(lxmax+0.1,0.0);
move2(0.0,lymin-0.1);
draw2(0.0,lymax+0.1);
*/

/* color(YELLOW); */
move2(lxmin+sc_x*(x[0]-xmin),lymin+sc_y*(y[0]-ymin));
for(i=1;i<=no;++i) draw2(lxmin+sc_x*(x[i-1]-xmin),lymin+sc_y*(y[i-1]-ymin));

/* swapbuffers(); */
}


float max_array(x,no)
float *x;
int no;
{
 int i;
 float maxv;

 maxv=x[0];
 for(i=2;i<=no;++i) { if(x[i-1]>maxv) maxv=x[i-1];}
 return(maxv);
}

float min_array(x,no)
float *x;
int no;
{
 int i;
 float minv;

 minv=x[0];
 for(i=2;i<=no;++i) { if(x[i-1]<minv) minv-x[i-1];}
 return(minv);
}


/*************************************************************************/
/*     plots the steering angle and drive force for the path             */
/*************************************************************************/
plot steer_and_drive_forces(int_increment, cell-no)
int Tnt_increment;
int cell_no;
{
 long  cell-id; /*cell id*/
 int   i;


 /* cell-no = 7*/
 cell-id = winpar.wingid[cell_no-1];
 winset(cell_id);
 color(BLACK);
 clear();

 linewidth(1);
 color(YELLOW);
 move(tout2[0],force.beta[0],0.0);
 /* for(i=2;i<=npoints2;++i) draw(tout2[i-1],force.beta[i-1],0.0); */
 for(i=1;i<=npoints2;i=i+int_increment) draw(tout2[i-1],force.beta[i-1],0.0);

 /*draw limits on Beta*/
 color(CYAN);
 move(  0.0,-PI/3.0,0.0);
```

```
 draw(150.0, -PI/3.0,0.0);
 move(  0.0,PI/3.0,0.0);
 draw(150.0,PI/3.0,0.0);

 color(RED);
 move(0.0,0.0,0.0);
 draw(150.0,0.0,0.0);
 move(0.0,-5.0,0.0);
 draw(0.0,5.0,0.0);

 swapbuffers();

 /* draw drive force in window no 8*/
 cell-id = winpar.wingid[cell_no];
 winset(cell_id);
 color(BLACK);
 clear();

 color(YELLOW);
 move(tout2[0],force.drive[0],0.0);
 /*for(i=2;i<=npoints2;++i) draw(tout2[i-1],force.drive[i-1],0.0); */
 for(i=1;i<=npoints2;i=i+int_increment) draw(tout2[i-1],force.drive[i-1],0.0);

 color(RED);
 move(0.0,0.0,0.0);
 draw(150.0,0.0,0.0);
 move(0.0, -6500.0,0.0);
 draw(0.0,6500.0,0.0);

 /*draw limits on Ft*/
 color(CYAN);
 /* 3-16-95
 move(  0.0,-0.3*1550*9.81,0.0);
 draw(120.0, -0.3*1550*9.81,0.0);
 move(  0.0,  0.3*1550*9.81,0.0);
 draw(120.0,  0.3*1550*9.81,0.0);
 */

 move(  0.0,-6000.0,0.0);
 draw(150.0, -6000.0,0.0);
 move(  0.0,  3000.0,0.0); /*old limit 2000*/
 draw(150.0,  3000.0,0.0);


 swapbuffers();
}


/*-------------generates ellipse-----------------------------------*/
void ellipsef(x, y, a, b)
float x, y, a, b;
{
int    i;
int    npoint = 100:
float theta = 0.0;
float delta:
float cp[100][2];
float cpx[100], cpy[100];

delta = 2*PI/(npoint-1);
/*printf("delta = %f\n",delta); */
for (i = 0;i<npoint;++i)
I
  cp[i][0] = x + a*cos(theta) ;
  cpx[i]   = cp[i][0];
  cp[i][1] = Y + b*sin(theta) ;
```

```
  cpy[i]   = cp[i][1];
  theta = theta + delta;
  /*printf("cp(%d) = %f, %f\n", i, cpx[i], cpy[i]); */
}
move(cpx[0],cpy[0],0.0);
for(i=2;i<=npoint;++i) draw(cpx[i-1],cpy[i-1],0.0);
}
/*-------------------------------------------------------------------*/
```

```
/*****************************************************************************
written by Satish
computes the steering angle at each point along the path using date obtained
from the runge-kutta integration
*****************************************************************************/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "runge.h"
#include "kinematic.h"

#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define PI              3.141592654
/***********critical slip angles for the front and rear tires*************?****/
#define  ALPHA_F_CR             10*PI/180 /* 10 degrees */
#define  ALPHA_R_CR             10*PI/180 /* 10 degrees */
/*****************************************************************************/
/**********major and minor axes for friction ellipse******************/
#define  A_fr_ellipse           6000.0
#define  B_fr_ellipse           80000.0*ALPHA_R_CR
/*****************************************************************************/

extern struct p {int winposi[4][10];/*screen coords of the windows*/
                 float wincoord[6][10];/* limits on the local coordinate
                                          systems of the windows*/
                 int wingid[10]; /*window ids*/
                 int nwin;  /*total number of windows opened*/
                                } winpar ;

extern struct vehicle {float mass, Ic, d, alpha, alpha f, alpha r, Ft, Beta,
                       fnr,  fnf, width, wheel-length, wheel-width; } car;


extern float s[500],x[500][2],xs[500][2],xss[500][2];
extern int PATH NO;
extern struct paths {float theta[500],theta_s[500],theta_ss[500];} path;
extern struct paths2 {float alpha[500],alpha_s[500],alpha_ss[500];} pdyn;
extern struct forces {float beta[1000], drive[1000], dis[1000]; int np;} force;

/*for the solution of the steering angle*/
float A_eq, B_eq;

float psi-f; /*angle between the velocity of front wheel and the car body
              x - axis; SEE NOTES*/
float rhs_of_force_eq; /*right hand side of the y-force equation with the
                         Beta term on the left hand side*/

extern int store-flag-for-runge-2; /*for storing data*/
extern float Beta-max, Beta-min;  /*limits on the steering angle*/

extern int existence-flag; /* 0 if roots exists, 1 if they do not*/

extern int violation-of-wheel-slip; /* 0 if wheel slip angle limits are not
                                       violated,  1 if they are*/

extern int violation_of_fr_ellipse; /* 0 if friction ellipse is not violated,
                                       1 if it is*/

/*****************************************************************************/
/*program to find the steering angles at different points along the path*/
/*****************************************************************************/
float ALPHA-REAR, ALPHA-FRONT; /*rear and front tyre slip angles*/
```

```
find_path_steer_angles(int_increment)
int int_increment;
{
FILE *fout;

/*files for debugging*/
/*
FILE *dbg_sdd, *dbg_ts;
*/

float solve steering_angle();
float solve_drive_force();
float get_quantity_at_s();

float sd-at-s, sdd-at-s, theta-at-s, thetas-at-s, thetass_at_s;
float alpha-at_s, alphas-at-s;
float Cfront; 7*force constant for the front steering wheel*/
float Crear:

int   i;

/*to ensure that drive force lies in dugoff ellipse*/
float Fxmax, Fymax;

/*initialize existence flag*/
existence-flag = 0;

/*initialize violation-of-wheel-slip*/
violation-of-wheel-slip = 0;

/*initialize violation_of_fr_ellipse*/
violation_of_fr_ellipse = 0;

/* number of points */
force.np = npoints2;


/*open file for storing sdd*/
  /*
  if(store_flag_for_runge_2==1) dbg_sdd = fopen("dbg_sdd.mat","w");
  */

/*open file for storing theta-s = K*/
  /*
  if (store_flag_for_runge_2==1) dbg_ts = fopen("dbg_ts.mat","w");
  */

/*for (i=1;i<=PATH_NO;++i)*/
/*for (i-l; i<=npoints2;++i)*/
for (i=1;i<=npoints2;i=i+int_increment)
{
  if(store_flag_for-runge_2==1) printf ("s = %f\n", tout2[i-1]);
  force.dis[i-1] = tout2[i-1];

  sd_at_s       = get-quantity-at-s("sdv",tout2[i-11);
  if(store_flag_for_runge_2==1) printf("sd = %f\n", sd-at-s);

  sdd at s      = get_quantity_at_s("sdd",tout2[i-1]);
  if (store-flag-f or_runge_2==1) printf("sdd = %f\n", sdd-at-s);

  /*store the accelerations for debugging*/
  /*
  if(store_flag_for_runge_2==1) fprintf(dbg_sdd,"%f\n",sdd_at_s);
  */

  theta-at-s    = get_quantity_at_s("the",tout2[i-1]); /*path.theta[i-1];*/
```

```
  if(store_flag_for_runge_2==1) printf ("theta = %f\n", theta-at-s);

  thetas-at-s   = get-quantity-at s("ths",tout2[i-1]); /*path.theta_s[i-1] */
  if(store_flag_for_runge_2==1) printf("thetas = %f\n", thetas-at-s);

  /*store  theta-s = K for debugging*/
  /*
  if(store_flag_for_runge_2==1) fprintf(dbg_ts,"%f\n",thetas_at_s);
  */

  alpha-at-s    = yout2[i-1][0]; /*pdyn.alpha[i-1];*/
  if(store_flag_for_runge_2==1) printf("alpha = %f\n", alpha-at-s);

  alphas-at-s   = yout2[i-1][1]; /*pdyn.alpha_s[i-1];*/
  if(store_flag_for_runge_2==1) printf("alphas = %f\n", alphas-at-s);

  Cfront = -80000.0;
  Crear  = 80000.0;

  /*find the steering angle at each point using a golden search*/
  force.beta[i-1] = solve_steering_angle(tout2[i-1],sd_at_s,sdd_at_s,
                                         alpha-at-s,  alphas-at-s,
                                         theta-at-s,  thetas-at-s, Crear);

  /*find the driving force required*/
  force.drive[i-1] = solve_drive_force(tout2[i-1],sd_at_s,sdd_at_s,
                                         alpha-at-s,  alphas-at-s,
                                         theta_at_s,  thetas-at-s, Cfront,
                                         force.beta[i-1]);

  if(store_flag_for_runge_2==1) printf("beta = %f\n", force.beta [i-1]);

  /*****for  plots******/
  tyre.alpha_r[i-1] = 180.0*ALPHA_REAR/PI;
  tyre.alpha_f[i-1] = 180.0*ALPHA_FRONT/PI;

  /*****check violation of wheel slip angles******/
  if (
       (  abs(tyre.alpha_r[i-1])  > ALPHA_R CR*180/PI )
                    ||
       (  abs(tyre.alpha_f[i-1])  > ALPHA_F CR*180/PI )
     )
     violation-of-wheel-slip  = 1;


  /*****check violation of friction ellipse******/
  find_ellipse(tyre.alpha_r[i-1]*PI/180.0,&Fxmax,&Fymax);
       /*( sqr(force.drive[i-1]/A_fr_ellipse)
                    +
         sqr(Crear*tyre.alpha_r[i-1]*PI/B_fr_ellipse/180.0) )*/
  if (
        ( sqr(force.drive[i-1]/Fxmax)
                    +
         sqr(Crear*tyre.alpha_r[i-1]*PI/B_fr_ellipse/180.0) )
                    >     1.0
     )
     violation_of_fr_ellipse = 1;

  /*getchar();*/
}

  /*fclose(dbg_sdd);*/ /*close acceleration debug file*/
  /*fclose(dbg_ts);*/ /*close theta-s debug file*/

if (store_flag_for_runge_2==1)
{
```

```
  /*store the force and steering angles for plotting*/
  fout = fopen("steer.mat","w");
   for (i=1;i<=npoints2;i=i+int_increment) fprintf(fout,"%f\n",force.beta[i-1]*180.(
  fclose(fout);

  fout = fopen("drive.mat","w");
   for (i=1;i<=npoints2;i=i+int_increment) fprintf(fout,"%f\n",force.drive[i-1]);
  fclose(fout);

  fout = fopen ("pdis.mat","w");
   for (i=1;i<=npoints2;i=i+int_increment) fprintf(fout,"%f\n",tout2[i-1]);
  fclose(fout);

  fout = fopen("alpha_front.mat","w");
                  /* note:alpha_f is negative of actual value*/
   for (i=1;i<=npoints2;i=i+int_increment) fprintf(fout,"%f\n",-tyre.alpha_f[i-1]);
  fclose(fout);

  fout = fopen("alpha_rear.mat","w");
   for (i=1;i<=npoints2;i=i+int_increment) fprintf(fout,"%f\n",tyre.alpha_r[i-1]);
  fclose(fout);

  fout = fopen("fr_ellipse.mat","w");
                    /*stores the value of (ft/a)**2+(fr/b)**2*/
   for (i=1;i<=npoints2;i=i+int_increment) fprintf(fout,"%f\n",
       sqr(force.drive[i-1]/A_fr_ellipse) +
          sqr(Crear*tyre.alpha_r[i-1]*PI/B_fr_ellipse/180.0) );
  fclose(fout);
}

}

/*************************************************************************/
/*   solves for the driving force at a given point                       */
/*************************************************************************/
float solve_drive_force(dis,sd_dis,sdd_dis,alpha_dis,alphas_dis,theta_dis,
thetas_dis, Cr, steer-angle)
float dis,sd_dis,sdd_dis,alpha_dis,alphas_dis,theta_dis,thetas_dis;
float Cr,  steer-angle;
{
float f_fn(), anglex();

float alpha-front:
float drive-force-at-dis:
float kappa-at-dis;
float psi-front;

psi-front = anglex( cos(alpha_dis),
                car.d*(thetas_dis+alphas_dis) - sin(alpha_dis) );

alpha-front = steer-angle - psi-front:

/** for plots**/
ALPHA-FRONT = alpha-front;

kappa-at-dis    = thetas-dis;

drive-force-at-dis = car.mass*(sdd_dis*cos(alpha_dis) +
                     kappa_at_dis*sqr(sd_dis)*sin(alpha_dis) ) +
                     f_fn(alpha_front,Cr)*sin(steer_angle);

return(drive_force_at_dis);
}

/*************************************************************************/
/*   solves for the steering angle at a given point                      */
```

```
/*****************************************************************************/
float solve_steering_angle(dis,sd_dis,sdd_dis,alpha_dis,alphas_dis,theta_dis,
thetas_dis, Cr)
float dis,sd_dis,sdd_dis,alpha_dis,alphas_dis,theta_dis,thetas_dis;
float Cr;
{
float f_alpha_r(), f_fn(), anglex(), gold-find-steer-angle0;
float kappa-at-dis;
float alpha-rear;

/*printf("dis=%f\n", dis); */

alpha-rear = f_alpha_r(dis, sd_dis, alpha-dis, alphas_dis*sd_dis, thetas_dis*sd_dis

/**for plots**/
ALPHA-REAR = alpha-rear;

psi_f = anglex( cos(alpha_dis),
                car.d*(thetas_dis+alphas_dis) - sin(alpha_dis) );

kappa-at-dis     = thetas-dis;

rhs_of_force_eq = car.mass*(-sdd_dis*sin(alpha_dis) +
                  kappa_at_dis*sqr(sd_dis)*cos(alpha_dis)) -
                  f_fn(alpha_rear, Cr);

/* solve the equation of the form:
      f_nf(Beta-psi_f)*cos(Beta) - rhs_of_force-eq = 0
  =>  -Cr*abs(Beta-psi_f)*sgn(sin(Beta-psi_f))*cos(Beta) = rhs_of_force_eq
  Refer notes for details                                  */

/*first check if a solution does NOT exist. The conditions for this are:

        Beta_min > psi-f - |rhs_of_force_eq/Cr|
and     Beta_max < psi-f + |rhs_of_force_eq/Cr|          */

if (
    ( Beta_min > psi-f - abs(rhs_of_force_eq/Cr) )
                        &&
    ( Beta-max < psi-f + abs(rhs_of_force_eq/Cr) )
   )
{
   printf("SOLUTION DOES NOT EXIST AT S = %f\n", dis);
   existence-flag = 1;
}

return( gold_find_steer_angle() );
}

float gold_find_steer_angle()
{
 float equation_cost(), minval();
 int   get_root_index();

 float xl, fl, xu, fu;
 float optv, optx;
 float optv1, optxl, optv2, optx2, optv3, optx3;
 float xmid1, xmid2;
 float diff[3];

 float y:
 float cost, min_cost;
 float min_y;
 float min_y_1, min_y_2;

 int   ret_index;
```

```
 int   1:
 int   N;
 int   flag-for-root3;

 N  = 50:
/*use the golden search method to locate one minimum*/
 xl = 0.0;
 fl = equation_cost(xl);
 xu = PI/3.0;
 fu = equation_cost(xu);
 gsearch(xl,fl,xu,fu,N,  &optv1,&min_y_1) ;

 xl = -PI/3.0;
 fl = equation_cost(xl);
 xu = 0.01;
 fu = equation_cost(xu);
 gsearch(xl,fl,xu,fu,N,   &optv2,&min_y_2) ;

 /*added Feb 27 95 to check existence */
 if ( (abs(optv2) > 0.1) && (abs(optv1) > 0.1) )
 {
    printf("solution may not exist! !!\n");
    existence-flag = 1;
 }

 if( optv2 < optvl) return(min_y_2);
 else               return(min_y_1);


 min_cost = 10000000.0;
/*check exhaustivelyfor thecost*/
 for (y= 0.0; y< PI/3-0.1; y = y+0.02)
 {

  cost = equation-cost(y);
  printf("cost(%f)=%f\n",y,cost);
  if (cost < min-cost)
  {
    min_cost = cost;
    min_y = y;
  }
  /*
  getchar();
  */
 }
 for (y= -0.02; y> -PI/3+0.1; y = y-0.02)
 {

  cost = equation-cost(y);
  printf("cost(%f)=%f\n",y,cost);
  if (cost < min-cost)
  {
    min_cost = cost:
    min_y = y;
  }
  /*
  getchar();
  */
 }
 return(min_y);

/* solve for the first root*/
 xl = -PI;
 fl = equation_cost(xl);
 xu = PI:
 fu = equation_cost(xu);
```

```
 gsearch(xl,fl,xu,fu,N,  &optv1,&optx1) ;
 printf("optv1=%f, optxl = %f\n", optvl, optxl);
 if (abs(optv1)>0.01) printf("no roots, ERROR\n");

/*check the interval left of optxl excluding the point optxl */
 xl = -PI;
 fl = equation_cost(xl);
 xu = optxl -(optx1-PI)/200.0;
 fu = equation_cost(xu);
 gsearch(xl,fl,xu,fu,N,   &optv,&optx) ;

if ( abs(optv) < 0.001 )
{
  optx2  = optx;
  optv2  = optv;
}
else
{
/* check the interval to the right of optxl */
  xl = optxl + (PI-optx1)/200;
  fl = equation_cost(xl);
  xu = PI;
  fu = equation_cost(xu);
  gsearch(xl,fl,xu,fu,N,  &optv2,&optx2);
}

/*check is there is more than one root*/
if ( abs(optv2) > 0.001 ){ printf("there is only one root\n"); return(optx1);}

if ( optx2 < optxl )
{
 xmidl = optx2;
 xmid2 = optxl;
}
else
{
 xmidl = optxl;
 xmid2 = optx2;
}

flag_for_root3 = 0; /*flag to indicate whether there is a third root*/
for ( i=1;i<=3;++i)
{
    if (i==1)
    {
       xl = -PI;
       xu = xmidl;
    }
    else if ( i==2 )
    {
       xl = xmidl;
       xu = xmid2;
    }
    else
    {
       xl = xmid2;
       xu = PI:
    }

    fl = equation_cost(xl);
    fu = equation-cost(xu);
    gsearch(xl,fl,xu,fu,N,   &optv,&optx) ;
    if ( abs(optv) < 0.01)
    {
      flag-for-root3 = 1;
      if ( (abs(optx-optxl) > 0.01) && (abs(optx-optx2) > 0.01) )
```

```
      {
         optx3  = optx;
         optv3  = optv;
      }
    }
}
if (flag-for-root3 == 0)
{
  printf("there are only two roots\n");
  /*return the root closer to zero*/
  diff[0] = minval( optxl, abs(2*PI-optx1) );
  diff[1] = minval( optx2, abs(2*PI-optx2) );
  diff[2] = 1000.0;
  ret-index = get_root_index(diff,3);
}
else
{
  /*of the three roots, return the one closer to 0*/
  diff[0] = abs(optx1); /*minval( optxl, abs(2*PI-optx1) );*/
  diff[1] = abs(optx2); /*minval( optx2, abs(2*PI-optx2) );*/
  diff[2] = abs(optx3); /*minval( optx3, abs(2*PI-optx3) );*/
  ret-index = get_root_index(diff,3);
}
  if (ret_index==1) return (optxl);
  if (ret-index==2) return(optx2);
  if (ret index==3) return(optx3);
}

int get_root_index(diff)
float diff[3];
{
if ( (diff[0] <= diff[l]) && (diff[0] <= diff[2]) ) return(1);

if ( (diff[1] <= diff[0]) && (diff[1] <= diff[2]) )  return(2);

if ( (diff[2] <= diff[0]) && (diff[2] <= diff[1]) )  return(3);
}

float minval(val1,val2)
float val1, val2;
{
if (val1 <= va12) return(val1);
else              return(va12);
I

float  equation-cost(angle)
float angle;
{
float f_fn();
float Cfront; /*force constant for the front wheel*/

Cfront = -80000;

/*
printf("in equation cost\n");
printf("at %f, cost = %f\n", x,abs( f_fn(x-A_eq,Cfront)*cos(x) - B_eq ));
*/
return( abs( f_fn(angle-psi_f,Cfront)*cos(angle) - rhs_of_force_eq )/10000.0 );
}

/*************************************************************************/
/*minimizes a function of one variable using the golden search method   */
/*************************************************************************/
gsearch(xl,fl,xu,fu,N,       optv,optx)
float xl, fl, xu, fu; /*inputs*/
float *optv,*optx;   /*outputs*/
```

```
{
float equation_cost();

float xl, fl, x2, f2;
float tau;
int   k;

tau = 0.381936;

xl = (1-tau)*xl + tau*xu;
fl = equation_cost(x1);
x2 = tau*xl + (1-tau)*xu;
f2 = equation-cost(x2);


k = 3;
k = k+1;

while (  k < N   )
{
  if (  fl > f2 )
  {
     xl = xl;
     fl = fl,!
     xl = x2;
     fl = f2;
     x2 = tau*xl + (1-tau)*xu;
     f2 = equation_cost(x2);
  }
  else
  I
     xu = x2;
     fu = f2;
     x2 = xl;
     f2 = fl;
     xl = (1-tau)*xl + tau*xu;
     fl = equation_cost(x1);
  }
  k = k+1;
}

*optx = xl;
*optv = equation_cost(x1);
}




float gold_find_steer_angle_2()
{
 float equation_cost(), minval();
 int   get_root_index();

 float xl, fl, xu, fu;
 float n interval: /*number of intervals*/
 float interval-size;
 float opt_f[20]; /*optimum function values for each interval*/
 float opt_x[20]; /*optimum beta     values for each interval*/

 float ROOT1, F_ROOT1; /*the root closest to zero*/
 float ROOT2, F_ROOT2; /*other root*/
 int  i:
 int  N;

 int   index of root1, index_of_root2;
```

```
/*************************************************************************/
/*    There are atmost 2 roots (see notes for details)                   */
/*    divide the interval [-pi/2, pi/2] into several intervals and       */
/*    use the golden search method in those intervals to find the        */
/*    roots                                                              */
/*************************************************************************/

/*divide [-pi/2,pi/2] into n-interval intervals*/
 n_interval    = 10;
 interval-size  =  PI/n-interval;

 N  = 50;
 for (i=1;i<=n_interval;++i)
 {
   xl = -PI/2 + (i-1)*interval_size;
   fl = equation cost(xl);
   xu  =  xl  + -interval_size;
   fu = equation_cost(xu);
   gsearch(xl,fl,xu,fu,N,  &opt_f[i-1],&opt_x[i-1]) ;

 }


/*search for the two best roots*/
 ROOT1   = 0.0;
 F_ROOT1 = 10000000;
 for (i=1;i<=n_interval;++i)
   if(opt_f[i-1] < F_ROOT1)
      { index of_root1 = i; F_ROOT1 = opt_f[i-1]; ROOT1 = opt_x[i-1]; }

 ROOT2   = 0.0;
 F_ROOT2 = 10000000;
 for (i=1;i<=n interval;++i)
   if(  (opt_f[i-1] < F_ROOT2) && (i!=index_of_root1) )
      { index-of-root2 = i; F-ROOT2 = opt_f[i-1]; ROOT2 = opt_x[i-1]; }

if ( (abs(F_ROOT1) > 0.1) && (abs(F_ROOT2) > 0.1) )
{
   printf("solution may not exist!! !\n");
   existence-flag = 1;
}

/*Return the root closest to zero*/
if ( abs(ROOT1) <= abs(ROOT2) ) return(ROOT1);
else                            return(ROOT2);
}
```

```
/*************************************************************************
written by Satish
computes alpha and alpha-s along the path using the data from the runge-kutta
integration
t************************************************************************/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#include "runge.h"

#define norm(x,y) sqrt(sqr(x) +sqr(y))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define  PI             3.141592654


extern float s[500],x[500][2],xs[500][2],xss[500][2];
extern int PATH-NO;
extern struct paths2 {float alpha[500],alpha_s[500],alpha_ss[500];} pdyn;

float  get-quantity-from-integ(qty,dist)
char  qty[3];
float dist;
{
  int i,  s index;
  float quantity;
  int   point-flag;

  point-flag = 0;
  /*find the indices of the points encompassing the given distance*/
  for (i=1;i<=npoints2-1;++i)
  {
    if ( (dist>=tout2[i-1]) && (dist<tout2[i]) ) {s-index = i;point_flag=1;}
  }

  if(point_flag==1)
  {
    if  ( (qty[0]=='a') && (qty[1]=='l') && (qty[2]=='p') )
    {
      /*if qty = "alp" return alpha*/
      quantity = yout2[s_index-1][0] +
      (dist-tout2[s_index-1])*(yout2[s_index][0]-yout2[s_index-1][0])/(tout2[s_inde:
      return(quantity);
    }
    else if  ( (qty[0]=='a') && (qty[1]=='l') && (qty[2]=='s') )
    {
      /*if qty = "als" return alpha*/
      quantity = yout2[s_index-1][1] +
      (dist-tout2[s_index-1])*(yout2[s_index][1]-yout2[s_index-1][1])/(tout2[s_inde
      return(quantity);
    }
    else
    {
      printf("quantity unrecognizable: ERROR!!! !\n");
      getchar();
      return(-100.0);
    }
  }
  else /*return all quantities at the end of the path*/
  {
    if  ( (qty[0]=='a') && (qty[1]=='l') && (qty[2]=='p') )
    {
      quantity = yout2[npoints2-1][0];
      return(quantity);
```

```
    }
    else if ( (qty[0]=='a') && (qty[1]=='l') && (qty[2]=='s') )
    {
      quantity = yout2[npoints2-1][1];
      return(quantity);
    }
    else
    {
      printf("quantity unrecognizable: ERROR! !! !\n");
      getchar();
      return(-100.0);
    }

  }


}


get_alpha_along_path()
{
   int i;
   float get-quantity-from-integ();

   printf("PATH_NO = %d\n",PATH_NO);
   printf ("at 1\n");
   getchar();
   for (i=1;i<=PATH_NO-2;++i)
   {
    printf("at %d in loop\n",i);
    pdyn.alpha[i-1] =  get-quantity-from-integ("alp",s(i-11);
    pdyn.alpha_s[i-1] = get-quantity-from-integ("als",s[i-11);
   }
   printf ("at 2\n") ;
   pdyn.alpha[PATH_NO-2] = pdyn.alpha[PATH_NO-3];
   pdyn.alpha[PATH_NO-1] = pdyn.alpha[PATH_NO-2];
   printf("at 3\n");
   pdyn.alpha_s[PATH_NO-2] = pdyn.alpha_s[PATH_NO-3];
   pdyn.alpha_s[PATH_NO-1] = pdyn.alpha_s[PATH_NO-2];
}
```

```
/*****************************************************************************
written by Satish
*****************************************************************************/
#include <gl.h>
#include <device.h>
#include <stdio.h>
#include <math.h>
#define norm(x,y) sqrt(sqr(x)+sqr(y))
#define  abs(x)          ( ((x)>=0.0) ? (x) : -(x) )
#define sqr(x) ((x)*(x))
#define PI              3.141592654
tdefine MAX-NPAR 30

extern struct tv {int nconv;  float conv[20][2];} bzvel_;
extern struct tw {int nconp; float conp[20][2];} bzw_;
extern float constant-velocity-of-car;

extern int vel_type;

extern char QTY[3]; /*to decide which to optimize*/

btopar (npar,par)
float par[MAX_NPAR];
int   *npar;
{
 int i;
 int nfirst, nlast;

 if      ( (QTY[0]=='v') && (QTY[1]=='e') && (QTY[2]=='l')  )
 {
   if (vel_type==0)
   {
     /*vary only the y-coordinates of the control points*/
     *npar = bzvel_.nconv;
     for (i=1;i<=bzvel_.nconv;++i)
     {
      par[i-1] = bzvel_.conv[i-1][1];
     }
   }
   else
   {
     *npar = bzvel_.nconv-1;
     for (i=2;i<=bzvel_.nconv;++i)
     {
      par[i-2] = bzvel_.conv[i-1][1];
     }
   }
 }
 else if       ( (QTY[0]=='e') && (QTY[1]=='n') && (QTY[2]=='v') )
 {
   if (vel_type==0)
   {
     /*vary the initial velocity and the y-coordinates of the control
       points for the acceleration profile*/
     *npar = bzvel_.nconv+1;
     par[0] = constant-velocity-of-car:
     for (i=1;i<=bzvel_.nconv;++i)
     {
      par[i] = bzvel_.conv[i-1][1];
     }
   }
   else
   {
     /* vary y-coordinates of the all the control points of the velocity
        profile  */
     *npar = bzvel_.nconv;
```

```
    for (i=1;i<=bzvel_.nconv;++i)
    {
     par[i-1] = bzvel_.conv[i-1][1];
    }
  }
}
else if ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='h') )
{
  nfirst = 1; /*3;*/ /*4;*/ /*3;*/
  nlast = 1;  /*3;*/ /*4;*/ /*3;*/
  *npar = 2*(bzw_.nconp-nfirst-nlast);
                  /*exclude the first nfirst and last nlast pts*/
  for (i=1;i<=bzw_.nconp-nfirst-nlast;++i)
  {
   par[2*(i-1)+0] = bzw_.conp[nfirst + i-1][0];
   par[2*(i-1)+1] = bzw_.conp[nfirst + i-l] [1];
  }
}
else if ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='2') )
{
/*fix the x coordinate of the second point and the y coordinate of the
  second last point*/
  *npar = 2*(bzw_.nconp-4) + 1 + 1;

  /*first parameter is the y coordinate of the 2nd control point*/
  par[0] = bzw_.conp[1][1];

  /*next  2*(bzw_.nconp-4) are the x and y coordinates of the points
    except for the last two points*/
  for (i=1;i<=bzw_.nconp-4;++i)
  {
   par[1+2*(i-1)+0] = bzw_.conp[2 + i-1][0];
   par [1+2*(i-1)+1] = bzw_.conp[2 + i-1][1];


  /*the next parameter is the x coordinate of the second last control point*/
   par[2*(bzw_.nconp-4)+1+1-l] = bzw_.conp [bzw_.nconp-l-l] [1]; /*for lane change*/

}
else if ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='3') )
{
/*fix the x coordinate of the second point and the y coordinate of the
  second last point*/
  *npar = 2" (bzw_.nconp-4) + 1 + 1    + 1; /*last parameter is the constant
                                              velocity of the car*/

  /*first parameter is the y coordinate of the 2nd control point*/
  par[0] = bzw_.conp[1][1];

  /*next 2*(bzw_.nconp-4) are the x and y coordinates of the points
    except for the last two points*/
  for (i=1;i<=bzw_.nconp-4;++i)
  {
   par[1+2*(i-1)+0] = bzw_.conp[2 + i-1][0];
   par[1+2*(i-1)+1] = bzw_.conp[2 + i-1][1];


  /*the next parameter is the x coordinate of the second last control point*/
   par[2*(bzw_.nconp-4)+1+1-1] = bzw_.conp[bzw_.nconp-1-1][1]; /*for lane change*/

   /*to determine the maximum velocity attainable*/
   par[2*(bzw_.nconp-4)+1+1+1-1] = constant-velocity-of-car:

}
else if ((QTY[0]=='T') && (QTY[1]=='R') && (QTY[2]=='J') )
{
```

```
  nfirst = 1: /*4; */
  nlast  = 1; /*4; */
  *npar = 2*(bzw_.nconp-nfirst-nlast);
                  /*exclude the first nfirst and last nlast pts*/
  for (i=1;i<=bzw_.nconp-nfirst-nlast;++i)
  {
   par[2*(i-1)+0] = bzw_.conp[nfirst + i-1][0];
   par[2*(i-1)+1] = bzw_.conp[nfirst + i-1][1];
  }

  if (vel_type==0)
  {
    /*vary only the y-coordinates of the control points*/
    *npar = *npar + bzvel_.nconv;
    for (i=1;i<=bzvel_.nconv;++i)
    {
     par[2*(bzw_.nconp-nfirst-nlast) + i-l] = bzvel_.conv[i-1][1];
    }
  }
  else
  {
    *npar = *npar + bzvel_.nconv-1;
    for (i=2;i<=bzvel_.nconv;++i)
    {
     par[2*(bzw_.nconp-nfirst-nlast)+ i-21 = bzvel_.conv[i-l][1];
    }
  }
}
else if ( (QTY[0]=='T') && (QTY[1]=='R') && (QTY[2]=='2') )
{
/*fix the x coordinate of the second point and the y coordinate of the
  second last point*/
  *npar = 2*(bzw_.nconp-4) + 1 + 1;

  /*first parameter is the y coordinate of the 2nd control point*/
  par[0] = bzw_.conp[1][1];

  /*next 2*(bzw_.nconp-4) are the x and y coordinates of the points
    except for the last two points*/
  for (i=1;i<=bzw_.nconp-4;++i)
  {
   par[1+2*(i-1)+0] = bzw_.conp[2 + i-1][0];
   par[1+2*(i-1)+1] = bzw_.conp[2 + i-1][1];
  }

  /*'the next parameter is the x coordinate of the second last control point*/
   par[2*(bzw_.nconp-4)+1+1-1] = bzw_.conp[bzw_.nconp-1-1][0];


  if (vel_type==0)
  {
    /*vary only the y-coordinates of the control points*/
    *npar = *npar + bzvel_.nconv;
    for (i=1;i<=bzvel_.nconv;++i)
    {
     par[2*(bzw_.nconp-4)+1+1+ i-l] = bzvel_.conv[i-1][1];
    }
  }
  else
  {
    *npar = *npar + bzvel_.nconv-1;
    for (i=2;i<=bzvel_.nconv;++i)
    {
     par[2*(bzw_.nconp-4)+1+1+ i-21 = bzvel_.conv[i-1][1];
    }
  }
```

```
 }
 else if      ((QTY[0]=='c') && (QTY[1]=='v') && (QTY[2]=='l') )
 /* optimize constant velocity*/
 {
   if (vel_type==1)

     printf("error in btopar, wrong velocity type\n"); getchar();
   }
   else
   {
     /* vary the constant velocity of the car*/
     *npar = 1;
      par[0] = constant-velocity-of-car:
   }
 }
 else
 {
   printf("error in optimization !! !\n"); getchar();
 }

}

partob(npar,par)
float par[MAX_NPAR];
int   npar;
{
 int i;
 int nfirst, nlast;

 if       ( (QTY[0]=='v') && (QTY[1]=='e') && (QTY[2]=='l') )
 {
   if(vel_type==0)
   {
     if (npar != bzvel_.nconv) {printf("ERROR in partob\n"); getchar();}
     for (i=1;i<=bzvel_.nconv;++i)
     {
      bzvel_.conv[i-1][1] = par[i-1] ;
     }
   }
   else
   I
     if (npar != bzvel_.nconv-1) {printf("ERROR in partob\n"); getchar();}
     for (i=2;i<=bzvel_.nconv;++i)
     {
      bzvel_.conv[i-1][1] = par[i-2] ;
     }
   }
 }
 else if       ( (QTY[0]=='e') && (QTY[1]=='n') && (QTY[2]=='v') )
 {
   if(vel_type==0)
   {
     if (npar != bzvel_.nconv+1) {printf ("ERROR in partob\n"); getchar();}
     constant_velocity-of-car = par[0];
     for (i=1;i<=bzvel_.nconv;++i)
     {
      bzvel_.conv[i-1][1] = par[i] ;
     }
   }
   else
   {
     if (npar != bzvel_.nconv) {printf ("ERROR in partob\n"); getchar();}
     for (i=1;i<=bzvel_.nconv;++i)
     {
      bzvel_.conv[i-1][1] = par[i-1] ;
     }
```

```
  }
}
else if ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='h') )
{
  nfirst = 1; /*3; */ /*4;*/ /*3;*/
  nlast  = 1; /*3; *//*4;*/ /*3;*/
  if (npar != 2*(bzw_.nconp-nfirst-nlast)) {printf("ERROR in partob\n"); getchar()
  for (i=1;i<=bzw_.nconp-nfirst-nlast;++i)
  {
   bzw_.conp[nfirst+ i-1][0] = par[2*(i-1)+0] ;
   bzw_.conp[nfirst+ i-l] [1] = par[2*(i-1)+1] ;
  }
}
else if ( (QTY[0]=='p') && (QTY[1]=='t') CC (QTY[2]=='2') )
{
  if( npar != 2*(bzw_.nconp-4)+1+1 )
        {printf("ERROR in partob\n"); getchar();}

  /*first parameter is the y coordinate of the 2nd control point*/
  bzw_.conp[2-1][1] = par[0];

  /*next 2*(bzw_.nconp-4) are the x and y coordinates of the points
    except for the last two points*/
  for (i=1;i<=bzw_.nconp-4;++i)

  'bzw_.conp[2 + i-1][0]= par[1+2*(i-1)+0];
   bzw_.conp[2 + i-1][1]= par[1+2*(i-1)+1];
  }

  /*the next parameter is the x coordinate of the second last control point*/
   bzw_.conp[bzw_.nconp-1-1][1]= par[2*(bzw_.nconp-4) +1+1-1]; /*for lane change*/

}
else if ( (QTY[0]=='p') && (QTY[1]=='t') && (QTY[2]=='3') )
{
  if( npar != 2*(bzw_.nconp-4)+1+1   +1 )
        {printf("ERROR in partob\n"); getchar();}

  /*first parameter is the y coordinate of the 2nd control point*/
  bzw_.conp[2-1][1] = par[0];

  /*next 2*(bzw_.nconp-4) are the x and y coordinates of the points
    except for the last two points*/
  for (i=1;i<=bzw_.nconp-4;++i)
  {
   bzw_.conp[2 + i-1][0]= par[1+2*(i-1)+0];
   bzw_.conp[2 + i-1][1]= par[1+2*(i-1)+1];
  }

  /*the next parameter is the x coordinate of the second last control point*/
   bzw_.conp[bzw_.nconp-1-1][1]= par[2*(bzw_.nconp-4)+1+1-l];  /*for lane change*/

   /*determine velocity of car*/
   constant-velocity-of-car = par[2*(bzw_.nconp-4)+1+1+1-l];
}
else if ( (QTY[0]=='T') && (QTY[1]=='R') && (QTY[2]=='J') )
{
  nfirst = 1; /*4;*/
  nlast  = 1; /*4;*/

  if (vel_type==0)
  {
    if(npar != 2*(bzw_.nconp-nfirst-nlast) + bzvel_.nconv)
                        {printf("ERROR in partob\n"); getchar();}

    for (i=1;i<=bzw_.nconp-nfirst-nlast;++i)
```

```
    {
     bzw_.conp[nfirst + i-l][0] = par[2*(i-1)+0];
     bzw_.conp[nfirst + i-l][1] = par[2*(i-1)+1];
    }

    for (i=1;i<=bzvel_.nconv;++i)
    {
     bzvel_.conv[i-1][1] = par[2*(bzw_.nconp-nfirst-nlast) + i-l];
    }
  }
  else
  {
    if(npar != 2*(bzw_.nconp-nfirst-nlast) + bzvel_.nconv - 1)
                          {printf("ERROR in partob\n"); getchar();}

    for (i=1;i<=bzw_.nconp-nfirst-nlast;++i)

     bzw_.conp[nfirst + i-l][0] = par[2*(i-1)+0];
     bzw_.conp[nfirst + i-l][1] = par[2*(i-1)+1];
    }

    for (i=2;i<=bzvel_.nconv;++i)
    {
     bzvel_.conv[i-1][1] = par[2*(bzw_.nconp-nfirst-nlast)+  i-21;
    }
  }
}
else  if  ( (QTY[0]=='T') && (QTY[1]=='R') && (QTY[2]=='2') )
{
  if (vel_type==0)
  {
    if(npar != 2*(bzw_.nconp-4)+1+1 + bzvel_.nconv)
                          {printf("ERROR in partob\n"); getchar();}

    /*first parameter is the y coordinate of the 2nd control point*/
    bzw_.conp[2-1][1] = par[0];

    /*next 2*(bzw_.nconp-4) are the x and y coordinates of the points
      except for the last two points*/
    for (i=1;i<=bzw_.nconp-4;++i)
    {
     bzw_.conp[2 + i-1][0]= par[1+2*(i-1)+0];
     bzw_.conp[2 + i-1][1]= par[1+2*(i-1)+1];
    }

    /*'the next parameter is the x coordinate of the second last control point*/
     bzw_.conp[bzw_.nconp-1-1][0]= par[2*(bzw_.nconp-4)+1+1-1];

    /*vary only the y-coordinates of the control points*/
    for (i=1;i<=bzvel_.nconv;++i)
     bzvel_.conv[i-1][1]= par[2*(bzw_.nconp-4)+1+1+ i-l];
  }
  else
  {
    if(npar != 2*(bzw_.nconp-4)+1+1 + bzvel_.nconv-1)
                          {printf("ERROR in partob\n"); getchar();}

    /*first parameter is the y coordinate of the 2nd control point*/
    bzw_.conp[2-1][1] = par[0];

    /*next 2*(bzw_.nconp-4) are the x and y coordinates of the points
      except for the last two points*/
    for (i=1;i<=bzw_.nconp-4;++i)
    {
     'bzw_.conp[2 + i-1][0]= par[1+2*(i-1)+0];
     bzw_.conp[2 + i-1][1]= par[1+2*(i-1)+1];
```

```
      }

      /*the next parameter is the x coordinate of the second last control point*/
       bzw_.conp[bzw_.nconp-1-1][0]= par[2*(bzw_.nconp-4)+1+1-l];

      /*for velocity profile*/
       for (i=2;i<=bzvel_.nconv;++i)
        bzvel_.conv[i-1][1]= par[2*(bzw_.nconp-4)+1+1+ i-21;

    }

 }
 else if      ( (QTY[0]=='c') && (QTY[1]=='v') && (QTY[2]=='l') )
 {
   if(vel_type==1)
   {
     printf("error in btopar, wrong velocity type\n"); getchar();
   }
   else
   {
     if (npar != 1) {printf("ERROR in partob\n"); getchar();}
     for (i=1;i<=bzvel_.nconv;++i)
     {
      constant-velocity-of-car = par[0] ;
     }
   }
 }
 else
 {
   printf("error in Optimization !!!\n"); getchar();
 }

}
```

```
C*****************************************************************************
c-----------------------------------------------------------------------------
c                  *** PATSH *** PATSH *** PATSH ***
        SUBROUTINE PATSH(PSI,SSI,PHI,THT,XFLG,N,DELS,DLMIN,
     1   ITLIM,IPT,NPITER,IPR,ICN,MERIT)
C UNCONSTRAINED FUNCTION MINIMIZATION:
C   MINIMIZE THE FUNCTION MERIT(X)
C
C ARGS:
C     PSI    (N) INPUT AS INITIAL GUESS TO SOLUTION
C            RETURNED AS THE FINAL SOLUTION
C     SSI    RETURNED AS FINAL FUNCTION VALUE
C     PHI    (N) TEMPOARY (TRIAL POINT)
C     THT    (N) TEMPORARY (PREVIOUS BASE)
C     XFLG   (N) TEMPORARY (VARIATIONS)
C     N      SIZE OF PROBLEM
C     DELS   INITIAL DELTA FOR VARIATIONS
C     DELMIN MINIMUM DELTA ALLOWED
C     ITLIM MAXIMUM NUMBER OF ITERATIONS ALLOWED
C     IPT    PRINT CONTROL
C            +1=DIAGNOSTIC PRINTOUT
C             0=MINIMAL PRINTOUT
c            -1=NO PRINTOUT
C     NPITER SPECIFIED PRINTING INTERVAL
C            EVERY NPITER'TH ITERATION IS PRINTED
C            NPITER=1 TO PRINT ALL ITERATIONS
C     IPR    PRINTER DEVICE
C     ICN    CONSOLE OUTPUT DEVICE
C            ICN CAN BE ZERO TO SUPPRESS PRINTING
C     MERIT EXTERNAL SUBROUTINE TO COMPUTE FUNCTION
C            CALLED AS MERIT(N,X,OBJ)
C            N IS SIZE OF PROBLEM
C            X(N) IS POINT TO EVALUATE FUNCTION
C            OBJ IS RETURNED AS FUNCTION VALUE
C*****************************************************************
C WRITTEN BY D.E. WHITNEY, MIT-JCF, 1974

c ... Added to abort optimization (9/6/88)
        common /abort/iabort

        DIMENSION PSI(5),PHI(5),THT(5),XFLG(5)
        integer*4 irand
        LOGICAL LWRIT
        external merit
C SELECTION OF ALFA DRASTICALLY AFFECTS CONVERGENCE!!
        DATA ALFA/1.02/
C.....FUNCTION F IS MINIMUM IMPROVEMENT REQUIRED OVER LAST BASEPOINT
        F(SSS)=SSS-ABS(SSS)*.0001*CUT
C
C PSI IS THE CURRENT BASEPT
C THT IS THE PREVIOUS BASEPT
C PHI IS THE TRIAL PT
C S IS THE OBJECTIVE FCT
C IPT= 1 FOR DIAGNOSTIC PRINTOUT
C      0 FOR MINIMAL OUTPUT
C     -1 FOR NO OUTPUT
C
C
C.....INITIALIZATION
        DEL=DELS
        LWRIT=.TRUE.
C LWRIT=.TRUE.  IF THIS ITERATION IS TO BE PRINTED
        IF (IPT.GE.0) WRITE(IPR,604) DEL,DLMIN,ITLIM,IPT
        DO 705 I=1,N
705     XFLG(I)=1.
```

```
        ITER=0
        irand = 1235431
        CUT=1.
C EVALUATE AT INIT BASEPT
10      CALL  MERIT(N,PSI,SSI)
        if(iabort.eq.1) go to 5000
90      SSITST=F(SSI)
C
C
C.....EXPLORE AROUND CURRENT BASEPOINT
100     S=SSI
        NPATM=0
        DO 101 I=1,N
101     PHI(I)=PSI(I)
        ICALL=1
        IF (IPT.LT.0.OR..NOT.LWRIT) GO TO 150
        WRITE(IPR,599) ITER
        WRITE(IPR,600) (PSI(J),J=1,N)
        WRITE(IPR,601) S,DEL
C MAKE EXPLORATORY MOVES
        GO TO 150
C IS PRESENT VALUE < BASEPT VALUE
160     IF(S.LT.SSITST) GO TO 200
c       ZIA=RAN (irand)    ! this was the original line
c RAND generates numbers in [0, 1.]  7-17-88
        ZIA=RANd(irand)
        IF(ZIA.ge. (1.-(.4)**(float(iter)))go to 200
C....CUT STEP SIZE
        DEL=DEL/2.
        IF(DEL.GT.DLMIN) GO TO 100
        IF(IPT.GE.0) WRITE(IPR,704)
        IF(CUT.LT.. 5) GO TO 702
C START OVER WITH INITIAL DEL AND CURRENT BASEPOINT
        IF(IPT.GE.0) WRITE(IPR,707)
        DEL=DELS
        CUT=0.
        GOT0 90
C
C
C.....SET NEW BASEPOINT.....MAKE PATTERN MOVE.....EXPLORE AROUND PATTERN
200     SSI=S
        SSITST=F(SSI)
        ITER=ITER+1
        NPATM=NPATM+1

        IF(ITER.GT.ITLIM) GO TO 700
        LWRIT=MOD(ITER,NPITER).EQ.0
        IF(IPT.LT.0.OR..NOT.LWRIT) GO TO 203
        WRITE(IPR,599) ITER
        WRITE(IPR,599) NPATM
        WRITE  (IPR,600) (PHI(I),I=1,N)
        WRITE (IPR,601) SSI,DEL
        IF(ICN.GT.0) WRITE(ICN,601) SSI,DEL
C MAKE PATTERN MOVE
203     DO 201 I=1,N
        THT(I)=PSI(I)
        PSI(I)=PHI(I)
201     PHI(I)=PHI(I)+ALFA*(PHI(I)-THT(I))
        CALL MERIT(N,PHI,SPI )
        if(iabort.eq.1) go to 5000
        S=SPI
        IF(IPT.NE.l.OR..NOT.LWRIT) GO TO 202 .
        WRITE(IPR,606) (PHI(I),I=1,N)
         WRITE(IPR,601) SPI,DEL
202     ICALL=2
C MAKE EXPL MOVES
```

```
        GO TO 150
C IS PRESENT VALUE < BASEPT VALUE
260     IF(S.LT.SSITST) GO TO 200
        GO TO 100
C
C
C . . . ..INTERNAL SUBROUTINE TO MAKE EXPLORATIONS ABOUT PHI
150     IWR-0
        IF(IPT.EQ.1.AND.LWRIT) IWR=1
        DO 180 K=1,N
        PHIOLD-PHI(K)
        STEPK=PHIOLD*.05
        IF(STEPK.EQ.0.) STEPK=.05
        STEPK=SIGN(STEPK*DEL,XFLG(K))
        PHI(K)-PHIOLD+STEPK
        CALL MERIT(N,PHI,SPI)
        if(iabort.eq.1) go to 5000
        IF(IWR.EQ.l) WRITE(IPR,602) ICALL,K,SPI,(PHI(L),L=1,N)
        IF(SPI.LT.S) GO TO 179
        XFLG(K)=-XFLG(K)
        PHI(K)=PHIOLD-STEPK
        CALL MERIT(N,PHI,SPI)
        if(iabort.eq.1) go to 5000
        IF(IWR.EQ.l) WRITE(IPR,602) ICALL,K,SPI,(PHI(L),L=1,N)
        IF (SPI.LT.S) GO TO 179
        PHI(K)=PHIOLD
        GO TO 180
179     S=SPI
180     CONTINUE
        GO TO (160,260),ICALL
C
C
700     IF (IPT.GE.0) WRITE(IPR,701)
702     DO 703 I=1,N
703     PSI(I)=PHI(I)
        IF(IPT.LT.0)  RETURN
        WRITE(IPR,607) ITER
        WRITE(IPR,708)
        WRITE(IPR,600) (PSI(I),I=1,N)
        WRITE(IPR,601) SSI,DEL
        RETURN
599     FORMAT(' **** ',I5)
600   FORMAT('    BASE PT:',1P,4(7E15.6,E15.6,/,T9))
606     FORMAT('  PATTERN:',1P,4(7E15.6,E15.6,/,T9))
601     FORMAT(6X,'OBJ:',1PE15.6,5X,'DEL:',E15.6)
602     FORMAT(1X,2I2,' OBJ: ',1PE14.6,'  TRIAL: ',4(6E14.6,E14.6,/,T35))
604     FORMAT('0DEL=',1PE15.6,',    DELMIN=',E15.6,0P,//,
     1 ' ITLIM=',I6,',   IPT=',I3)
607     FORMAT(' TOTAL NUMBER OF NEW BASEPOINTS (ITERATIONS) :',I5)
701     FORMAT('0SEARCH TERMINATED BECAUSE NUMBER OF ITERATIONS EXCEEDS LI
     1 MIT.')
704     FORMAT('0SEARCH TERMINATED BECAUSE STEPSIZE LESS THAN LIMIT.')
707     FORMAT('0 STARTING OVER WITH CURRENT BASEPOINT AND INITIAL DEL.')
708     FORMAT('  FINAL VALUES:')

5000    print  *, ' '
        print * , 'ABORTING  OPTIMIZATION'
        print*,'  '
        END
```

```
c 4-3-88 unix version ucla
c 2D 11-27-86
      subroutine bspline_w(jp, s, x, xs, xss,ier)
c=============================================================
c  Zvi Shiller 4-13-86 MIT.
c=============================================================
c      include 'resolution.par'
      dimension s(1000), x(2,1000),xs(2,1000),xss(2,1000)
      dimension xu(2,1000),xuu(2,1000)
      dimension r(4,2), r1(4,2), uv(4), q(4,4)
      real m(4,4), mr(4,2), kb(3), conp(2,20)
c passed to popt
c     integer*4 nconp
      common /bzw/ nconp, conp

      ier = 0
c point index
      jp = 0
      j2 = 0
      jspl = 0
c add one control point for a straight line:
c-----------------------------------11-27-86
      if(nconp.eq. 2) then
        conp(1,3) = conp(1,2)
        conp(2,3) = conp(2,2)
        conp(1,2) = (conp(1,3) + conp(1,1))/2
        conp(2,2) = (conp(2,3) + conp(2,1))/2
        nconp = 3
      endif

c number of splines
      nspl = nconp - 1
      if(nspl.lt.1) then
        ier=1
        write(6,*)' ier 1'
        return
      endif
c     70.               delta u (to have 70 points on the path)
cc      delu = nspl/70.
      delu = nspl/100.
      ju = 1./delu
      delu = 1./sngl(ju)
      uo = 0
c make sure that u=1
      ul = 1 + delu/10
c distance starts from 0
      s(1) = 0
      a0 = 0
c--------------------------------------
c  check distance between control points:
c--------------------------------------
      dist = 0
      do 1 j=1,nconp-1
        dis = 0
        do 2 i=1,2
          dis = dis + (conp(i,j+1)-conp(i,j))**2
2       continue
c accumulated distance 3-8-86
        dist = dist + sqrt(dis)
        if(dis. lt. .0005) then
          ier = 1
        write(6,*)' ier 2'
        endif
1     continue
```

```
        if(ier.eq.1) then
c initialize in case of error   3-8-86
          jp = 2
          s(jp) = dist
          return
        endif
c--------------------------------------
c  check distance to the first control point:  (no loops)
c--------------------------------------
        disl = 0
c skip testing distance
        ii=0
        if(ii.eq.0) go to 50
        do 3 j=2,nconp
          dis2 = 0
          do 4 i=1,2
            dis2 = dis2 + (conp(i,j)-conp(i,1))**2
         continue
          if(dis2. lt. disl) then
            ier = 1
          write(6,*)' ier 4'
c initialize in case of error   3-8-86
            jp = 2
            s(jp) = dist
            return
          endif
          disl = dis2
 3     continue
c-----------------------------------------------------------------
c  create the Q matrix (transforms the R matrix for each spline)
c-----------------------------------------------------------
50      do 5 i=1,4
        do 5 j=1,4
5       q(i,j) = 0

        q(1,2) = 1
        q(2,3) = 1
        q(3,4) = 1
c ---------------------
c  create the M matrix:  (this M matrix is for a B spline)
c-------------------------
        do 6 i=1,4
        do 6 j=1,4
        m(i,j) = 0

        m(1,1) = 1
        m(1,2) = 4
        m(1,3) = 1
        m(2,1) = -3
        m(2,2) = 0
        m(2,3) = 3
        m(3,1) = 3
        m(3,2) = -6
        m(3,3) = 3
        m(4,1) = -1
        m(4,2) = 3
        m(4,3) = -3
        m(4,4) = 1
        do 7 i=1,4
        do 7 j=1,4
7       m(i,j) = m(i,j)/6

c----------------------
c  create the first R matrix:
c-----------------------
c       create the first artificial point:  (to make spline pass through first
```

```
c       real control point)
        do 8 j=1,2
8       r(1,j) = 2*conp(j,1) - conp(j,2)

cfirst point on the path is also the first real conp
        do 9 i=2,4
        do 9 j=1,2
9       r(i, j) = conp(j,i-1)

c  multiply M*R
10      call mam4 (m,r,mr)
c increment spline index
        jspl = jspl + 1

c=========================
c  create points:  at each spline (defined by four control points) the
c parameter u goes from 0 to 1).
c-----------------------
        do 22 u = u0,u1,delu
c increment point index
          jp = jp + 1
          if(jp.eq.146) then
            jjj=1
          endif
c         ---------------------
c         compute x:
c         ----------------
          uv(1) = 1
          uv(2) = u
          uv(3) = u**2
          uv(4) = u**3
c compute point
          call vm4(uv,mr,x(1,jp))
c         ------------------------
c         compute dr/du:
c         ----------------------
          uv(1) = 0
          uv(2) = 1
          uv(3) = 2*u
          uv(4) = 3*u**2
c compute point
          jj = jp
          call vm4(uv,mr,xu(1,jj))
          uv(1) = 0
          uv(2) =  0
          uv(3) = 2
          uv(4) = 6*u
c compute second derivative
          call vm4(uv,mr,xuu(1,jj))
22     continue
c last point is the first point of next spline
        u0 = delu
c indecis for the last spline
        j1 = j2 + 1
        j2 = jp

c end of path
11      if(jspl. eq.  nspl) go to 20
c-----------------------------------
c  add one control point for the next spline
c  update the R matrix:
c-------------------------------
c nultiply Q*R = R1
        call mam4(q,r,r1)
c add next control point
        if(jspl. le. nspl-2) then
```

```
          r1(4,1) = conp(1,jspl+1+2)
          r1(4,2) = conp(2,jspl+1+2)
c   create last artificial control point
        else
          do 21 j=1,2
21        r1(4,j) = 2*conp(j,nconp) - conp(j,nconp-1)

        endif
c   replace rl by r:
        do 12 i=1,4
        do 12 j=1,2
12      r(i,j) = r1(i,j)

        go to 10

c---------------------------
c   compute S, Xs, Xss:
c---------------------------
20      do 13 j=1,jp-1
          bb = 0
          aa = 0
          do 14 i=1,2
            bb = bb + (x(i,j+1) - x(i,j))**2
            aa = aa + xu(i,j)**2
14        continue
          if(aa. It. 1E-5) aa = .00001
          dsu = sqrt(aa)
          do 15 i=1,2
15        xs(i,j) = xu(i,j)/dsu

          ds = sqrt(bb)
          s(j+1) = s(j) + ds
c   compute kB = xs x xuu/aa:
          kb(1) = 0
          kb(2) = 0
          kb(3) = (xs(1,j)*xuu(2,j) - xs(2,j)*xuu(1,j))/aa
c   compute xss = kB x xs:
          xss(1,j) = - kb(3)*xs(2,j)
          xss(2,j) = kb(3)*xs(1,j)

13      continue
c   take care of last point:
        do 16 i=1,2
          xss(i,jp) = xss(i,jp-1)
          xs(i,jp) = xs(i,jp-1)
16      continue

        return
        end
```

```
c 4-3-88 unix version ucla
c 2D 11-27-86
      subroutine bspline_vel(jp, s, x, xs, xss,ier)
c=======================================================================
c  Zvi Shiller 4-13-86 MIT.
c=======================================================================
c      include 'resolution.par'
      dimension s(1000), x(2,1000),xs(2,1000),xss(2,1000)
      dimension xu(2,1000),xuu(2,1000)
      dimension  r(4,2), r1(4,2), uv(4), q(4,4)
      real  m(4,4), mr(4,2), kb(3), conv(2,20)
c passed to popt
c     integer*4 nconv
      common /bzvel/ nconv, conv

      ier = 0
c point index
      jp = 0
      j2 = 0
      jspl = 0
c add one control point for a straight line:
c---------------------------------------11-27-86
      if(nconv. eq. 2) then
        conv(1,3) = conv(1,2)
        conv(2,3) = conv(2,2)
        conv(1,2) = (conv(1,3) + conv(1,1))/2
        conv(2,2) = (conv(2,3) + conv(2,1))/2
        nconv = 3
      endif

c number of splines
      nspl = nconv - 1
      if(nspl.lt.1) then
        ier=1
        write(6,*)' ier 1'
        return
      endif
c     70.               delta u (to have 70 points on the path)
cc      delu = nspl/70.
      delu = nspl/300.
      ju = 1./delu
      delu = 1./sngl(ju)
      uo = 0
c make sure that u=1
      ul = 1 + delu/10
c distance starts from 0
      s(1) = 0
      a0 = 0
c--------------------------------------
c  check distance between control points:
c--------------------------------------
      dist = 0
      do 1 j=1,nconv-1
        dis = 0
        do 2 i=1,2
          dis = dis + (conv(i,j+1)-conv(i,j))**2
2       continue
c accumulated distance 3-8-86
        dist = dist + sqrt(dis)
        if(dis. lt. .0005)  then
          ier = 1
        write(6,*)' ier 2'
        endif
1     continue
```

```
        if(ier.eq.1) then
c initialize in case of error   3-8-86
          jp = 2
          s(jp) = dist
          return
        endif
c--------------------------------------
c  check distance to the first control point: (no loops)
c--------------------------------------
        disl = 0
c skip testing distance
        ii=0
        if(ii.eq.0) go to 50
        do 3 j=2,nconv
          dis2 = 0
          do 4 i=1,2
            dis2 = dis2 + (conv(i,j)-conv(i,1))**2
4         continue
          if(dis2. lt. disl) then
            ier = 1
          write(6,*)' ier 4'
c initialize in case of error   3-8-86
            jp = 2
            s(jp) = dist
            return
          endif
          disl = dis2
 3     continue
c-------------------------------------------------------------
c  create the Q matrix (transforms the R matrix for each spline)
c-------------------------------------------------------------
50      do 5 i=1,4
        do 5 j=1,4
5       q(i,j) = 0

        q(1,2) = 1
        q(2,3) = 1
        q(3,4) = 1
c----------------------
c  create the M matrix:  (this M matrix is for a B spline)
c-------------------------
        do 6 i=1,4
        do 6 j=1,4
6       m(i,j) = 0

        m(1,1) = 1
        m(1,2) = 4
        m(1,3) = 1
        m(2,1) = -3
        m(2,2) = 0
        m(2,3) = 3
        m(3,1) = 3
        m(3,2) = -6
        m(3,3) = 3
        m(4,1) = -1
        m(4,2) = 3
        m(4,3) = -3
        m(4,4) = 1
        do 7 i=1,4
        do 7 j=1,4
7       m(i,j) = m(i,j)/6

c - - - - - - - - - - - - - - - m - - - - -
c  create the first R matrix:
c------------------------
c       create the first artificial point:  (to make spline pass through first
```

```
c       real control point)
        do 8 j=1,2
8       r(1,j) = 2*conv(j,1) - conv(j,2)

c first point on the path is also the first real conp
        do 9 i=2,4
        do 9 j-1,2
9       r(i,j) = conv(j,i-1)

c  multiply M*R
10      call mam4(m,r,mr)
c increment spline index
        jspl = jspl + 1

c=========================
c  create points:  at each spline (defined by four control points) the
c parameter u goes from 0 to 1).
c--------------------
        do 22 u = u0,u1,delu
c increment point index
          jp = jp + 1
          if(jp.eq.146) then
            jjj=1
          endif
c         ---------------------
c         compute x:
c         ----------------
          uv(1) = 1
          uv(2) = u
          uv(3) = u**2
          uv(4) = u**3
c compute point
          call vm4 (uv,mr,x (1,jp))
c         ------------------------
c         compute dr/du:
c         --------------------
          uv(1) = 0
          uv(2) = 1
          uv(3) = 2*u
          uv(4)  = 3*u**2
c compute point
          jj = jp
          call vm4(uv,mr,xu(1,jj))
          uv(1) = 0
          uv(2) = 0
          uv(3) = 2
          uv(4) = 6*u
c compute second derivative
          call vm4(uv,mr,xuu(1,jj))
22     continue
c last point is the first point of next spline
        u0 = delu
c indecis for the last spline
        j1 = j2 + 1
        j2 = jp

c end of path
11      if(jspl. eq.  nspl) go to 20
c-------------------------------m-----
c  add one control point for the next spline
c  update  the  R matrix:
c------------------------------
c nultiply Q*R = R1
        call mam4(q,r,r1)
c add next control point
        if(jspl. le. nspl-2) then
```

```
          r1(4,1) = conv(1,jspl+1+2)
          r1(4,2) = conv(2,jspl+1+2)
c  create last artificial control point
        else
          do 21 j=1,2
21        r1(4,j) = 2*conv(j,nconv) - conv(j,nconv-1)

        endif
c  replace rl by r:
        do 12 i=1,4
        do 12 j=1,2
12      r(i,j) = r1(i,j)

        go to 10

c---------------------------
c  compute S, Xs, Xss:
c-----------------"---------
20      do 13 j=1,jp-1
          bb = 0
          aa = 0
          do 14 i=1,2
            bb = bb + (x(i,j+l) - x(i,j))**2
            aa = aa + xu(i,j)**2
14        continue
          if(aa. lt. 1E-5) aa = .00001
          dsu = sqrt(aa)
          do 15 i=1,2
15        xs(i,j) = xu(i,j)/dsu

          ds = sqrt(bb)
          s(j+1) = s(j) + ds
c  compute kB = xs x xuu/aa:
          kb(1) = 0
          kb(2) = 0
          kb(3) = (xs(1,j)*xuu(2,j) - xs(2,j)*xuu(1,j))/aa
c  compute xss = kB x xs:
          xss(1,j) = - kb(3)*xs(2,j)
          xss(2,j) = kb(3)*xs(1,j)

13      continue
c  take care of last point:
        do 16 i=1,2
          xss(i,jp) = xss(i,jp-1)
          xs(i,jp) = xs(i,jp-1)
16      continue

        return
        end
```

```
c 4-3-88 unix version ucla
c 2D 11-27-86
        subroutine bspline(jp, s, x, xs, xss,ier)
c=======================================================================
c  Zvi Shiller 4-13-86 MIT.
c=======================================================================
c        include 'resolution.par'
        dimension s(1000), x(2,1000),xs(2,1000),xss(2,1000)
        dimension xu(2,1000),xuu(2,1000)
        dimension  r(4,2), r1(4,2), uv(4), q(4,4)
        real  m(4,4), mr(4,2), kb(3), conp(2,20)
c passed to popt
c       integer*4 nconp
        common /bz/ nconp, conp

        ier = 0
c point index
        jp = 0
        j2 = 0
        jspl = 0
c add one control point for a straight line:
c---------------------------------------11-27-86
        if(nconp. eq. 2) then
          conp(1,3) = conp(1,2)
          conp(2,3) = conp(2,2)
          conp(1, 2) = (conp(1,3) + conp(1,1))/2
          conp(2,2) = (conp(2,3) + conp(2,1))/2
          nconp = 3
        endif

c number of splines
        nspl = nconp - 1
        if(nspl.lt.1) then
          ier=1
          write(6,*)' ier 1'
          return
        endif
c       70.               delta u (to have 70 points on the path)
cc        delu = nspl/70.
        delu = nspl/300.
        ju = 1./delu
        delu = 1./sngl(ju)
        uo = 0
c make sure that u-l
        ul = 1 + delu/10
c distance starts from 0
        s(1) = 0
        a0 = 0
c--------------------------------------
c  check distance between control points:
c--------------------------------------
        dist = 0
        do 1 j=1,nconp-1
          dis = 0
          do 2 i=1,2
            dis = dis + (conp(i,j+1)-conp(i,j))**2
2         continue
c accumulated distance 3-8-86
          dist = dist + sqrt(dis)
          if(dis. lt..0005) then
            ier = 1
          write(6,*)' ier 2'
          endif
1       continue
        if(ier.eq.1) then
```

```
c initialize in case of error   3-8-86
          jp = 2
          s(jp) = dist
          return
        endif
c-------------------------------------
c  check distance to the first control point:  (no loops)
c-------------------------------------
        disl = 0
c skip testing distance
        ii-0
        if(ii.eq.O) go to 50
        do 3 j=2,nconp
          dis2 = 0
          do 4 i=1,2
            dis2 = dis2 + (conp(i,j)-conp(i,1))**2
4         continue
          if(dis2. lt. disl) then
            ier = 1
          write(6,*)' ier 4'
c initialize in case of error   3-8-86
            jp = 2
            s(jp) = dist
            return
          endif
          disl = dis2
 3     continue
c-------------------------------------------------------------
c  create the Q matrix (transforms the R matrix for each spline)
c-------------------------------------------------------------
50      do 5 i=1,4
        do 5 j=1,4
5       q(i,j) = 0

        q(1,2) = 1
        q(2,3) = 1
        q(3,4) = 1
c----------------------
c  create the M matrix: (this M matrix is for a B spline)
c-------------------------
        do 6 i=1,4
        do 6 j=1,4
6       m(i,j) = 0

        m(1,1) = 1
        m(1,2) = 4
        m(1,3) = 1
        m(2,1) = -3
        m(2,2) = 0
        m(2,3) = 3
        m(3,1) = 3
        m(3,2) = -6
        m(3,3) = 3
        m(4,1) = -1
        m(4,2) = 3
        m(4,3) = -3
        m(4,4) = 1
        do 7 i=1,4
        do 7 j=1,4
7       m(i,j) = m(i,j)/6

c----------------------
c  create the first R matrix:
c--------------------a--
c       create the first artificial point:  (to-make spline pass through first
c       real control point)
```

```
        do 8 j=1,2
8       r(1,j) = 2*conp(j,1) - conp(j,2)

c first point on the path is also the first real conp
        do 9 i-2,4
        do 9 j=1,2
9       r(i,j) = conp(j,i-1)

c  multiply M*R
10      call mam4 (m,r,mr)
c increment spline index
        jspl = jspl + 1

c=========================
c  create points:  at each spline (defined by four control points) the
c parameter u goes from 0 to 1).
c--------------------
        do 22 u = u0,u1,delu
c increment point index
          jp = jp + 1
          if(jp.eq.146) then
            jjj=1
          endif
c         ---------------------
c         compute x:
c         ----------------
          uv(1) = 1
          uv(2) = u
          uv(3) = u**2
          uv(4) = u**3
c compute point
          call vm4(uv,mr,x(1,jp))
c         ------------------------
c         compute dr/du:
c         --------------------
          uv(1) = 0
          uv(2) = 1
          uv(3) = 2*u
          uv(4) = 3*u**2
c compute point
          jj = jp
          call vm4(uv,mr,xu(1,jj))
          uv(1) = 0
          uv(2) = 0
          uv(3) = 2
          uv(4) = 6*u
c compute second derivative
          call vm4(uv,mr,xuu(1,jj))
22     continue
c last point is the first point of next spline
        u0 = delu
c indecis for the last spline
        j1 = j2 + 1
        j2 = jp

c end of path
11      if(jspl. eq.  nspl) go to 20
c-----------------------------------
c  add one control point for the next spline
c  update the R matrix:
c-------------------------------
c nultiply Q*R = R1
        call mam4(q,r,r1)
c add next control point
        if(jspl. le. nspl-2) then
          r1(4,1) = conp(1,jspl+1+2)
```

```
          r1(4,2) = conp(2,jspl+1+2)
c  create last artificial control point
        else
          do 21 j=1,2
21        r1(4,j) = 2*conp(j,nconp) - conp(j,nconp-1)

        endif
c  replace rl by r:
        do 12 i-1,4
        do 12 j=1,2
12      r(i,j) = r1(i,j)

        go to 10

c--------------------------
c  compute S, Xs, Xss:
c--------------------------
20      do 13 j=1,jp-1
          bb = 0
          aa = 0
          do 14 i=1,2
            bb = bb + (x(i,j+l) - x(i,j))**2
            aa = aa + xu(i,j)**2
14        continue
          if(aa. lt. 1E-5) aa = .00001
          dsu = sqrt(aa)
          do 15 i=1,2
15        xs(i,j) = xu(i,j)/dsu

          ds = sqrt(bb)
          s(j+1) = s(j) + ds
c  compute kB = xs x xuu/aa:
          kb(1) = 0
          kb(2) = 0
          kb(3) = (xs(1,j)*xuu(2,j) - xs(2,j)*xuu(1,j))/aa
c  compute xss = kB x xs:
          xss(1,j) = - kb(3)*xs(2,j)
          xss (2,j) = kb(3)*xs(1,j)

13     continue
c  take care of last point:
        do 16 i=1,2
          xss (i,jp) = xss(i,jp-1)
          xs(i,jp) = xs(i,jp-1)
16     cont inue

        return
        end
```

```
        subroutine mam4(am1,am2,a12)
c-----------------------------------
c  mam4 multiplys 4x4 by 4x2 matrices:
c-----------------------------------
        dimension am1(4,4),am2(4,2),a12(4,2)
        do 20 i=1,4
        do 20 j=1,2
        a=0
        do 10 k=1,4
10      a=am1(i,k)*am2(k,j)+a
        a12(i,j)=a
20      continue
        return
        end


        subroutine vm4(v,am,av)
c-----------------------------------
c  vm4 multiplys vector*matrix = vector
c-----------------------------------
        dimension v(4), am(4,2), av(2)
        do 20 i=1,2
        a=0
        do 10 j=1,4
10      a = a + v(j) * am(j,i)
        av(i) = a
20      continue
        return
        end
```

```
c 4-3-88 unix version ucla
c 2D 11-27-86
      subroutine bspline_w2(jp, s, x, xs, xss,ier)
c=======================================================================
C  Zvi Shiller 4-13-86 MIT.
c=======================================================================
C       include 'resolution.par'
      dimension s(1000), x(2,1000),xs(2,1000),xss(2,1000)
      dimension xu(2,1000),xuu(2,1000)
      dimension r(4,2), r1(4,2), uv(4), q(4,4)
      real m(4,4), mr(4,2), kb(3), conp(2,20)
c passed to popt
C     integer*4 nconp
      common /bzw/ nconp, conp

      ier = 0
c point index
      jp = 0
      j2 = 0
      jspl = 0
c add one control point for a straight line:
c---------------------------------------11-27-86
      if(nconp. eq. 2) then
        conp(1,3) = conp(1,2)
        conp(2,3) = conp(2,2)
        conp(1,2) = (conp(1,3) + conp(1,1))/2
        conp(2,2) = (conp(2,3) + conp(2,1))/2
        nconp = 3
      endif

c number of splines
      nspl = nconp - 1
      if(nspl.lt.1) then
        ier=1
        write(6,*)' ier 1'
        return
      endif
C     70.               delta u (to have 70 points on the path)
cc      delu = nspl/70.
      delu = nspl/100.
      ju = 1./delu
      delu = 1./sngl(ju)
      uo = 0
c make sure that u-l
      u1 = 1 + delu/10
c distance starts from 0
      s(1) = 0
      a0 = 0
c--------------------------------------
C  check distance between control points:
c--------------------------------------
      dist = 0
      do 1 j=1,nconp-1
        dis = 0
        do 2 i=1,2
          dis = dis + (conp(i,j+1)-conp(i,j))**2
2       continue
c accumulated distance 3-8-86
        dist = dist + sqrt(dis)
        if(dis. lt. .0005) then
          ier = 1
        write(6,*)' ier 2'
        endif
1     cont inue
```

```
        if(ier.eq.1) then
c initialize in case of error   3-8-86
          jp = 2
          s(jp) = dist
          return
        endif
c---------------------------------------
c  check distance to the first control point: (no loops)
c----------------------_---------------
        disl = 0
c skip testing distance
        ii=0
        if(ii.eq.0) go to 50
        do 3 j=2,nconp
          dis2 = 0
          do 4 i=1,2
            dis2 = dis2 + (conp(i,j)-conp(i,1))**2
4         continue
          if(dis2. lt. disl) then
            ier = 1
          write(6,*)' ier 4'
c initialize in case of error   3-8-86
            jp = 2
            s(jp) = dist
            return
          endif
          disl = dis2
 3      continue
c---------------------------------------------------------------
c  create the Q matrix (transforms the R matrix for each spline)
c---------------------------------------------------------------
50      do 5 i=1,4
        do 5 j=1,4
5       q(i,j) =  0

        q(1,2) = 1
        q(2,3) = 1
        q(3,4) = 1
c---------------------
c  create the M matrix:  (this M matrix is for a B spline)
c-------------------------
        do 6 i=1,4
        do 6 j=1,4
6       m(i,j) = 0

        m(1,1) = 1
        m(1,2) = 4
        m(1,3) = 1
        m(2,1) = -3
        m(2,2) = 0
        m(2,3) = 3
        m(3,1) = 3
        m(3,2) = -6
        m(3,3) = 3
        m(4,1) = -1
        m(4,2) = 3
        m(4,3) = -3
        m(4,4) = 1
        do 7 i=1,4
        do 7 j=1,4
7       m(i,j) = m(i,j)/6

c----------------------
c  create the first R matrix:
c-----------------------
c       create the first artificial point:  (to make spline pass through first
```

```
c       real control point)
        do 8 j=1,2
8       r(1,j) = 2*conp(j,1) - conp(j,2)

c first point on the path is also the first real conp
        do 9 i=2,4
        do 9 j=1,2
9       r(i,j) = conp(j,i-1)

c  multiply M*R
10      call mam4 (m,r,mr)
c increment spline index
        jspl = jspl + 1

c=========================
c  create points:  at each spline (defined by four control points) the
c parameter u goes from 0 to 1).
c--------------------
        do 22 u = u0,u1,delu
c increment point index
          jp = jp + 1
          if(jp.eq.146) then
            jjj=1
          endif
c         ---------------------
c         compute x:
c         ----------------
          uv(1) = 1
          uv(2) = u
          uv(3) = u**2
          uv(4) = u**3
c compute point
          call vm4 (uv,mr,x(1,jp))
c         ------------------------
c         compute dr/du:
c         --------------------
          uv(1) = 0
          uv(2) = 1
          uv(3)  = 2*u
          uv(4) = 3*u**2
c compute point
          jj = jp
          call vm4 (uv,mr,xu(1,jj))
          uv(1) = 0
          uv(2) = 0
          uv(3) = 2
          uv(4) = 6*u
c compute second derivative
          call vm4(uv,mr,xuu(1,jj))
22     continue
c last point is the first point of next spline
        u o = delu
c indecis for the last spline
        j1 = j2 + 1
        j2 = jp

c end of path
11      if(jspl. eq. nspl) go to 20
c-----------------------------------
c  add one control point for the next spline
c  update the R matrix:
c-----------------------------
c nultiply Q*R = R1
        call mam4 (q,r,r1)
c add next control point
        if(jspl. le. nspl-2) then
```

```
          r1(4,1) = conp(1,jspl+1+2)
          r1(4,2) = conp(2,jspl+1+2)
c  create last artificial control point
        else
          do 21 j=1,2
21        r1(4,j) = 2*conp(j,nconp) - conp(j,nconp-1)

        endif
c  replace r1 by r:
        do 12 i=1,4
        do 12 j-1,2
12      r(i,j) = r1(i,j)

        go to 10

c--------------------------
c  compute S, Xs, Xss:
c--------------------------
20      do 13 j=1,jp-1
          bb = 0
          aa = 0
          do 14 i=1,2
            bb = bb + (x(i,j+1) - x(i,j))**2
            aa = aa + xu(i,j)**2
14        continue
          if(aa. lt. 1E-5) aa = .00001
          dsu = sqrt(aa)
          do 15 i=1,2
15        xs(i, j) = xu(i,j)/dsu

          ds = sqrt(bb)
          s(j+1) = s(j) + ds
c  compute kB = xs x xuu/aa:
          kb(1) = 0
          kb(2) = 0
          kb(3) = (xs(1,j)*xuu(2,j) - xs(2,j)*xuu(1,j))/aa
c  compute xss = kB x xs:
          xss(1,j) = - kb(3)*xs(2,j)
          xss(2, j) = kb(3)*xs(1,j)

13     continue
c  take care of last point:
        do 16 i=1,2
          xss(i,jp) = xss(i,jp-1)
          xs(i,jp) = xs(i,jp-1)
16     continue

        return
        end
```

```
c 4-3-08 unix version ucla
c 2D 11-27-86
        subroutine bspline_vel2(jp, s, x, xs, xss,ier)
c=======================================================================
c  Zvi Shiller 4-13-86 MIT.
c=======================================================================
c        include 'resolution.par'
        dimension s(1000), x(2,1000),xs(2,1000),xss(2,1000)
        dimension xu(2,1000),xuu(2,1000)
        dimension  r(4,2), r1(4,2), uv(4), q(4,4)
        real  m(4,4),mr(4,2), kb(3), conv(2,20)
c  passed  to  popt
c       integer*4 nconv
        common /bzvel/ nconv, conv

        ier = 0
c point index
        jp = 0
        j2 = 0
        jspl = 0
c add one control point for a straight line:
c-------------------------------------11-27-86
        if(nconv. eq. 2) then
          conv(1,3) = conv(1,2)
          conv(2,3) = conv(2,2)
          conv(1,2) = (conv(1,3) + conv(1,1))/2
          conv(2,2) = (conv(2,3) + conv(2,1))/2
          nconv = 3
        endif

c number of splines
        nspl = nconv - 1
        if(nspl.lt.1) then
          ier=1
          write(6,*)' ier 1'
          return
        endif
c       70.               delta u (to have 70 points on the path)
cc        delu = nspl/70.
        delu = nspl/300.
        ju = 1./delu
        delu = 1./sngl(ju)
        uo = 0
c make sure that u=1
        ul = 1 + delu/10
c distance starts from 0
        s(1) = 0
        a0 = 0
c-------------------------------------
c  check distance between control points:
c-------------------------------------
        dist = 0
        do 1 j=1,nconv-1
          dis = 0
          do 2 i=1,2
            dis = dis + (conv(i,j+1)-conv(i,j))**2
2         continue
c accumulated distance 3-8-86
          dist = dist + sqrt(dis)
          if(dis. lt..0005) then
            ier = 1
          write(6,*)' ier 2'
          endif
1       continue
```

```
        if(ier.eq.1) then
c initialize in case of error   3-8-86
          jp = 2
          s(jp) = dist
          return
        endif
c--------------------------------------
c  check distance to the first control point: (no loops)
c--------------------------------------
        disl = 0
c skip testing distance
        ii-0
        if(ii.eq.O) go to 50
        do 3 j-2,nconv
          dis2 = 0
          do 4 i=1,2
            dis2 = dis2 + (conv(i,j)-conv(i,1))**2
4         continue
          if(dis2. lt. disl) then
            ier = 1
          write(6,*)' ier 4'
c initialize in case of error   3-8-86
            jp = 2
            s(jp) = dist
            return
          endif
          disl = dis2
 3     continue
c-------------------------------------------------------------
c  create the Q matrix (transforms the R matrix for each spline)
c-------------------------------------------------------------
50      do 5 i=1,4
        do 5 j=1,4
5       q(i,j) = 0

        q(1,2) = 1
        q(2,3) = 1
        q(3,4) = 1
c---------------------
c  create the M matrix:  (this M matrix is for a B spline)
C----------W-------------
        do 6 i=1,4
        do 6 j=1,4
6       m(i,j) = 0

        m(1,1) = 1
        m(1,2) = 4
        m(1,3) = 1
        m(2,1) = -3
        m(2,2) = 0
        m(2,3) = 3
        m(3,1) = 3
        m(3,2) = -6
        m(3,3) = 3
        m(4,1) = -1
        m(4,2) = 3
        m(4,3) = -3
        m(4,4) = 1
        do 7 i=1,4
        do 7 j=1,4
7       m(i, j) = m(i,j)/6

c---------------------
c  create the first R matrix:
c-----------------------
c       create the first artificial point:  (to make spline pass through first
```

```
c       real control point)
        do 8 j=1,2
8       r(1,j) = 2*conv(j,1) - conv(j,2)

c first point on the path is also the first real conp
        do 9 i=2,4
        do 9 j=1,2
9       r(i,j) = conv(j,i-1)

c  multiply M*R
10      call mam4(m,r,mr)
c increment spline index
        jspl = jspl + 1

c=========================
c  create points:  at each spline (defined by four control points) the
c parameter u goes from 0 to 1).
c--------------------
        do 22 u = u0,u1,delu
c increment point index
          jp = jp + 1
          if(jp.eq.146) then
            jjj=1
          endif
c         ---------------------
c         compute x:
c         ----------------
          uv(1) = 1
          uv(2) = u
          uv(3) = u**2
          uv(4) = u**3
c compute point
          call vm4(uv,mr,x(1,jp))
c         ------------------------
c         compute dr/du:
c         --------------------
          uv(1) = 0
          uv(2) = 1
          uv(3) = 2*u
          uv(4) = 3*u**2
c compute point
          jj = jp
          call vm4(uv,mr,xu(1,jj))
          uv(1) = 0
          uv(2) = 0
          uv(3) = 2
          uv(4) = 6*u
c compute second derivative
          call vm4(uv,mr,xuu(1,jj))
22     continue
c last point is the first point of next spline
        u0 = delu
c indecis for the last spline
        j1 = j2 + 1
        j2 = jp

c end of path
11      if(jspl. eq. nspl) go to 20
c-------------------------------------
c  add one control point for the next spline
c  update the R matrix:
c------------------------------
c nultiply Q*R = R1
        call mam4 (q,r,rl)
c add next control point
        if(jspl. le. nspl-2) then
```

```
          r1(4,1) = conv(1,jspl+1+2)
          r1(4,2) = conv(2,jspl+1+2)
c   create last artificial control point
        else
          do 21 j=1,2
21        r1(4,j) = 2*conv(j,nconv) - conv(j,nconv-1)

        endif
c   replace rl by r:
        do 12 i=1,4
        do 12 j=1,2
12      r(i,j) = r1(i,j)

        go to 10

c---------------------------
c   compute S, Xs, Xss:
c---------------------------
20      do 13 j=1,jp-1
          bb = 0
          aa = 0
          do 14 i=1,2
            bb = bb + (x(i,j+1) - x(i,j))**2
            aa = aa + xu(i,j)**2
14        continue
          if(aa. It. 1E-5) aa = .00001
          dsu = sqrt(aa)
          do 15 i=1,2
15        xs(i, j) = xu(i,j) /dsu

          ds = sqrt(bb)
          s(j+1) = s(j) + ds
c   compute kB = xs X xuu/aa:
          kb(1) = 0
          kb(2) = 0
          kb(3) = (xs(1,j)*xuu(2,j) - xs(2,j)*xuu(1,j))/aa
c   compute xss = kB x xs:
          xss(l,j) = - kb(3)*xs(2,j)
          xss (2,j) = kb(3)*xs(1,j)

13      continue
c   take care of last point:
        do 16 i=1,2
          xss(i,jp) = xss(i,jp-1)
          xs(i,jp) = xs(i,jp-1)
16      continue

        return
        end
```

```
c 4-3-88 unix version ucla
c 2D 11-27-86
        subroutine bspline2(jp, s, x, xs, xss,ier)
c=======================================================================
c  Zvi Shiller 4-13-86 MIT.
c=======================================================================
c        include 'resolution.par'
        dimension s(1000), x(2,1000),xs(2,1000),xss(2,1000)
        dimension xu(2,1000),xuu(2,1000)
        dimension r(4,2), r1(4,2), uv(4), q(4,4)
        real m(4,4), mr(4,2), kb(3), conp(2,20)
c passed to popt
c       integer*4 nconp
        common /bz/ nconp, conp

        ier = 0
c point index
        jp = 0
        j2 = 0
        jspl = 0
c add one control point for a straight line:
c---------------------------------------11-27-86
        if(nconp. eq. 2) then
          conp(1,3) = conp(1,2)
          conp(2,3) = conp(2,2)
          conp(1,2) = (conp(1,3) + conp(1,1))/2
          conp(2,2) = (conp(2,3) + conp(2,1))/2
          nconp = 3
        endif

c number of splines
        nspl = nconp - 1
        if(nspl.lt.1) then
          ier=1
          write(6,*)' ier 1'
          return
        endif
c       70.               delta u (to have 70 points on the path)
cc        delu = nspl/70.
        delu = nspl/300.
        ju = 1./delu
        delu = 1./sngl(ju)
        uo = 0
c make sure that u=1
        ul = 1 + delu/10
c distance starts from 0
        s(1) = 0
        a0 = 0
c-------------------------------------
c  check distance between control points:
c-------------------------------------
        dist = 0
        do 1 j=1,nconp-1
          dis = 0
          do 2 i=1,2
            dis = dis + (conp(i,j+1)-conp(i,j))**2
2        continue
c accumulated distance 3-8-86
          dist = dist + sqrt(dis)
          if(dis. lt..0005) then
            ier = 1
          write(6,*)' ier 2'
          endif
       continue
        if(ier.eq.1) then
```

```
c initialize in case of error   3-8-86
          jp = 2
          s(jp) = dist
          return
        endif
c--------------------------------------
c  check distance to the first control point: (no loops)
c--------------------------------------
        disl = 0
c skip testing distance
        ii=0
        if(ii.eq.0) go to 50
        do 3 j=2,nconp
          dis2 = 0
          do 4 i=1,2
            dis2 = dis2 + (conp(i,j)-conp(i,1))**2
4         continue
          if(dis2. lt. disl) then
            ier = 1
          write(6,*)' ier 4'
c initialize in case of error   3-8-86
            jp = 2
            s(jp) = dist
            return
          endif
          disl = dis2
 3     continue
c-------------------------------------------------------------
c  create the Q matrix (transforms the R matrix for each spline)
c-------------------------------------------------------------
50      do 5 i=1,4
        do 5 j=1,4
5       q(i,j) = 0

        q(1,2) = 1
        q(2,3) = 1
        q(3,4) = 1
C----------------S----
c  create the M matrix:  (this M matrix is for a B spline)
c-------------------------
        do 6 i=1,4
        do 6 j=1,4
6       m(i,j) = 0

        m(1,1) = 1
        m(1,2) = 4
        m(1,3) = 1
        m(2,1) = -3
        m(2,2) = 0
        m(2,3) = 3
        m(3,1) = 3
        m(3,2) = -6
        m(3,3) = 3
        m(4,1) = -1
        m(4,2) = 3
        m(4,3) = -3
        m(4,4) = 1
        do 7 i=1,4
        do 7 j=1,4
7       m(i,j) = m(i,j)/6

c---------------------
c  create the first R matrix:
c-----------------------
c       create the first artificial point:  (to make spline pass through first
c       real control point)
```

```
        do 8 j=1,2
8       r(1,j) = 2*conp(j,1) - conp(j,2)

c first point on the path is also the first real conp
        do 9 i-2,4
        do 9 j=1,2
9       r(i,j) = conp(j,i-1)

c  multiply M*R
10      call mam4(m,r,mr)
c increment spline index
        jspl = jspl + 1

c=========================
c  create points:  at each spline (defined by four control points) the
c parameter u goes from 0 to 1).
c--------------------
        do 22 u = u0,u1,delu
c increment point index
          jp = jp + 1
          if(jp.eq.146) then
            jjj=1
          endif
c         _------m---w----- ----
c         compute x:
c         ----------------
          uv(1) = 1
          uv(2) = u
          uv(3) = u**2
          uv(4) = u**3
c compute point
          call vm4(uv,mr,x(1,jp))
c         ----------------------
c         compute dr/du:
c         --------------------
          uv(1) = 0
          uv(2) = 1
          uv(3) = 2*u
          uv(4) = 3*u**2
c compute point
          jj = jp
          call vm4(uv,mr,xu(1,jj))
          uv(1) = 0
          uv(2) = 0
          uv(3) = 2
          uv(4) = 6*u
c compute second derivative
          call vm4(uv,mr,xuu(1,jj))
22     continue
c last point is the first point of next spline
        u0 = delu
c indecis for the last spline
        j1 = j2 + 1
        j2 = jp

c end of path
11      if(jspl. eq. nspl) go to 20
c---------------------------------
c  add one control point for the next spline
c  update the R matrix:
c-------------------------------
c nultiply Q*R = R1
        call mam4(q,r,r1)
c add next control point
        if(jspl. le. nspl-2) then
          r1(4,1) = conp(1,jspl+1+2)
```

```
          rl(4,2) = conp(2,jspl+1+2)
c   create last artificial control point
        else
          do 21 j=1,2
21        rl(4, j) = 2*conp(j,nconp) - conp(j,nconp-1)

        endif
c   replace rl by r:
        do 12 i=1,4
        do 12 j=1,2
12      r(i, j) = rl(i,j)

        go to 10

c---------------------------
c   compute S, Xs, Xss:
c---------------------------
20      do 13 j=1,jp-1
          bb = 0
          aa = 0
          do 14 i=1,2
            bb = bb + (x(i,j+1) - x(i,j))**2
            aa = aa + xu(i,j)**2
14        continue
          if(aa. lt. 1E-5) aa = .00001
          dsu = sqrt(aa)
          do 15 i=1,2
15        xs(i, j) = xu(i,j)/dsu

          ds = sqrt(bb)
          s(j+1) = s(j) + ds
c   compute kB = xs x xuu/aa:
          kb(1) = 0
          kb(2) = 0
          kb(3) = (xs(1,j)*xuu(2,j) - xs(2,j)*xuu(1,j))/aa
c   compute xss = kB X xs:
          xss(1,j) = - kb(3)*xs(2,j)
          xss(2,j) = kb(3)*xs(1,j)

13      continue
c   take care of last point:
        do 16 i=1,2
          xss(i,jp) = xss(i,jp-1)
          xs(i,jp) = xs(i,jp-1)
16      continue

        return
        end
```