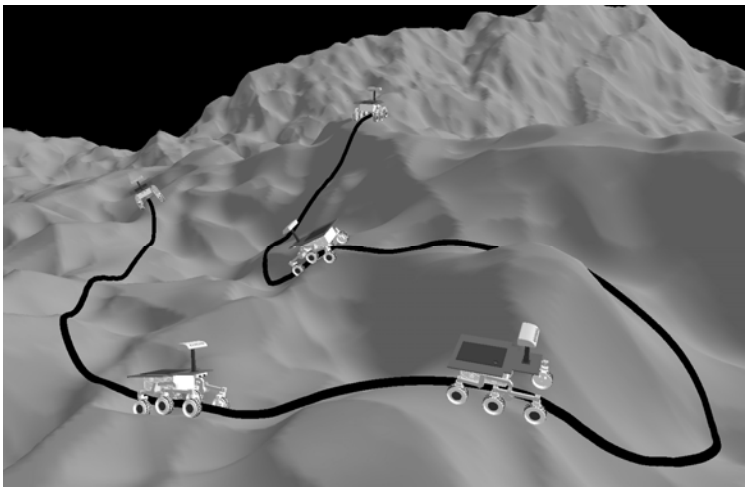
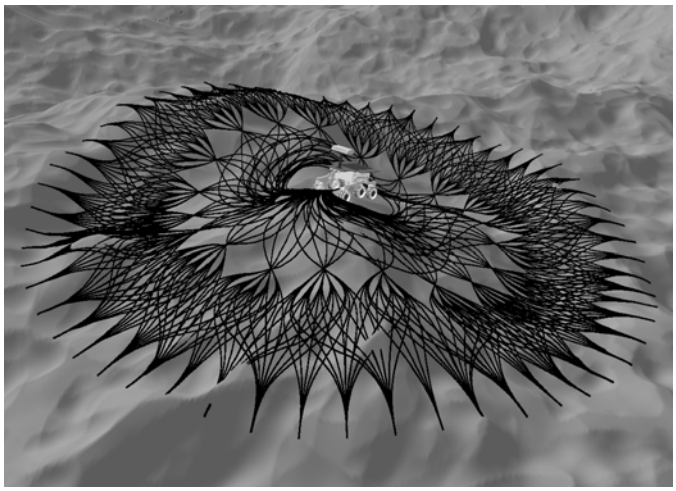
VERY ROUGH TERRAIN NONHOLONOMIC TRAJECTORY GENERATION AND MOTION PLANNING FOR PLANETARY ROVERS





TRAJECTORY GENERATION DOCUMENTATION

CONTACT(S): **Thomas M. Howard (thoward@cs.cmu.edu)**
Dr. Alonzo Kelly (alonzo@ri.cmu.edu)

REVISION DATE: **July 21st, 2008**

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Rough Terrain Trajectory Generation
DOCUMENT TYPE Trajectory Generator Schedule Document
AUTHOR Thomas M. Howard
REVISION DATE September 25th, 2006

PROJECT OBJECTIVE: A trajectory generation class shall be implemented to find trajectories for arbitrary vehicle models and arbitrary terrain which meet boundary position, heading, curvature, and velocity constraints. Trajectory generation problems of this type require numerical solution methods, because no closed-form solution is currently known. Therefore, this trajectory generation class will implement a numerical optimization method to minimize the boundary state error by adjusting the freedoms (parameterized controls) until convergence is met.

PROJECT SCHEDULE

Trajectory Generator	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
	CY 2004	CY 2005				CY 2006			
Visualization	■								
Test Environment		■							
Numerical Differentiation		■							
Rough Terrain Compensation			■						
Specific Rover Model (Rocky 7)				■					
Stage 1 CLARAty Integration				■					
Canonical Maneuvers					■				
Predictive Tracking Control						■			
CLARAty Integration Requirements and Concept Design						■			
Trajectory Generator Software Rewrite							■		
Trajectory Generator Software Testing								■	
Stage 2 CLARAty Integration									■
Trajectory Generator Detailed Design, User Guide, and Test Documents									■

PROJECT SCHEDULE DETAILS

Visualization (Q1): Develop visualization for trajectory generator system development.

Test Environment (Q2): Evaluate and/or develop simulation test environment used for trajectory generator system development.

Numerical Differentiation (Q2): Experiment with numerical linearization techniques used in the trajectory generator optimization

Rough Terrain Compensation (Q3): Develop the rough terrain trajectory generation algorithm. This requires generation of a suspension model and generalization of the forward simulation to three-dimensions.

Specific Rover Model (Rocky 7) (Q4): Develop a specific vehicle model applicable to the Mars Technology Program and test with the developed rough terrain trajectory generation technique.

Stage 1 CLARAty Integration (Q4): Integrate the first version of the trajectory generation software into CLARAty and perform field experiments to test the API and software robustness.

Canonical Maneuvers (Q5): Develop the capacity to generate a variety of maneuvers for vehicle mobility systems including forward and reverse corner-steering (Rocky 7), turn-in-place, skid-steering and all-wheel steering (Rocky 8).

Predictive Tracking Control (Q6): Develop a path tracking algorithm using the trajectory generator that predicatively compensates for terrain geometry, vehicle dynamics, and mobility system limitations.

CLARAty Integration Requirements and Concept Design (Q6): Develop and deliver the CLARAty integration requirements and concept design documents.

Trajectory Generator Software Rewrite (Q7): Rewrite the trajectory generator design and software for performance and robustness.

Trajectory Generator Software Testing (Q8): Test the rewritten trajectory generator software by measuring performance, convergence, memory requirements, etc….

Stage 2 CLARAty Integration (Q9): Integration the revised trajectory generator software into CLARAty and perform field experiments to validate the design.

Trajectory Generatro Detailed Design, User Guide, and Test Documents (Q9): Develop and Deliver the CLARAty integration detailed design, user guide, and test documents.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Rough Terrain Trajectory Generation
DOCUMENT TYPE Trajectory Generator Requirements Document
AUTHOR Thomas M. Howard
REVISION DATE September 25th, 2006

PROJECT OBJECTIVE: A trajectory generation class shall be implemented to find trajectories for arbitrary vehicle models and arbitrary terrain which meet boundary position, heading, curvature, and velocity constraints. Trajectory generation problems of this type require numerical solution methods, because no closed-form solution is currently known. Therefore, this trajectory generation class will implement a numerical optimization method to minimize the boundary state error by adjusting the freedoms (parameterized controls) until convergence is met.

PROJECT REQUIREMENTS

FUNCTIONAL REQUIREMENTS

- The trajectory generation algorithm shall generate a trajectory (parameterized control and path, represented as a sequence of vehicle states) given a pair of boundary state constraints, vehicle model, and vehicle controls parameterization. (R-1)
- The trajectory generator shall have an anytime trajectory generation mode for real-time operation of the algorithm. (R-2)

PERFORMANCE REQUIREMENTS

- The trajectory generator shall exhibit global convergence on flat terrain assuming the motion is feasible. (R-3)
- The trajectory generation algorithm shall exhibit 1% accuracy in position, heading, and curvature state constraints 99% of the time on average on rough terrain using the developed Rocky 8 Vehicle Model and the provided Generic Vehicle Model. (R-4)
- The trajectory generation algorithm shall exhibit average runtimes of less than 100 milliseconds in the situations described in requirement R-4. (R-5)
- The trajectory generation algorithm shall not have a memory footprint more than ten megabytes. (R-6)
- The trajectory generation algorithm shall not have a required initialization time more than one second. (R-7)

INTERFACE REQUIREMENTS

- The trajectory generator shall have an interface for trajectory generator that generates a trajectory (parameterized controls and path) given the parameterization of the vehicle controls and the initial and terminal boundary states. (R-8)
- The trajectory generator shall have an interface to an external vehicle model class for the modeling of the control dynamics, wheel/terrain interaction, vehicle dynamics, and suspension. (R-9)
- The trajectory generator shall have an interface to an external vehicle controls class for the representations of the parameterized body-frame vehicle controls. (R-10)
- The trajectory generator's vehicle model shall have an interface to an elevation map to allow for quasi-static modeling of the suspension. (R-11)
- The trajectory generator shall have an interface to a vehicle state class for the initialization of the forward simulation and representation of the system's state constraints. (R-12)

RELIABILITY REQUIREMENTS

- The trajectory generator shall have reasonableness checks in the system to detect implausible boundary state constraints. (R-13)
- The trajectory generator shall have reasonableness checks in the system to detect implausible parameterized control representations. (R-14)
- The trajectory generator shall have reasonableness checks in the system to detect implausible vehicle model representations. (R-15)
- The trajectory generator shall have reasonableness checks in the system to detect algorithm non-convergence and divergence. (R-16)
- The trajectory generator shall have runtime checks in the system to detect overrunning the time limit of the algorithm. (R-17)

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Rough Terrain Trajectory Generation
DOCUMENT TYPE Trajectory Generator Concept Design Document
AUTHOR Thomas M. Howard
REVISION DATE September 25th, 2006

INTRODUCTION: The trajectory generation algorithm can be decomposed into three basic functions: trajectory generation (numerical optimization), motion prediction (numerical integration), and vehicle modeling (vehicle simulation). The algorithm shall be designed so that the trajectory generation and the motion prediction remain vehicle independent, allowing the vehicle model to be easily swapped in and out for different applications (Figure 1). An overview of different aspects of this method will be covered in the concept design. Greater detail and examples of applications of the algorithm can be found in [1][2][3][4][5].

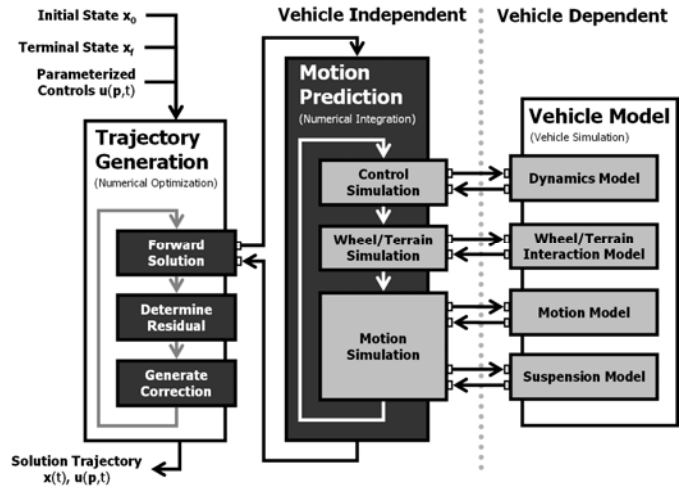


Figure 1: Trajectory Generator System Architecture. The trajectory generation algorithm can be conceptualized as a three-level hierarchy. The highest level is a numerical optimization that minimizes the constraint error by adjusting the free parameters in the controls. The next level, motion prediction, is simply the numerical integration of the equations of motion. The lowest level, the vehicle model, simulates the vehicle's dynamics, motion, and suspension. This architecture is formalized and implemented to permit different vehicle models to interface with the same underlying algorithm.

Trajectory Generation (Numerical Optimization): The numerical optimization step represents the highest level of the trajectory generation algorithm, where the difference between the target terminal state and the simulation terminal state is minimized by modifying the shape of the controls. In order to gain efficiency in the numerical optimization, the controls ($\mathbf{u}(\mathbf{p},t)$) shall be parameterized in a manner which nearly represents all feasible motions. The controls shall be parameterized in either linear and angular velocities or linear velocity and curvature and be functions of time or curvature as a function of arc length. A correction in the parameterized controls is computed using Newton's method, which requires a numerical estimate of the Jacobian of the system. Since the derivative of the differential equation which governs the system cannot be found in closed form, an estimate of the Jacobian shall be found by linearizing the system by each freedom in the parameterized control. The trajectory generation optimization will loop until the residual between the terminal states reaches an acceptable level or other termination conditions (divergence, max time, or max iterations) is observed.

The numerical method applied is Newton's method, where the Jacobian of the solution to the equations of motion is inverted to find a correction to the parameters in the system needed to minimize the constraint error ($\Delta\mathbf{x}_f(\mathbf{p})$), which we define for the moment to be the difference in the simulated terminal state and the terminal boundary constraints:

$$\left[\frac{\partial \Delta\mathbf{x}_\mathbf{f}(\mathbf{p})}{\partial \mathbf{p}}\right]\Delta\mathbf{p} = \left(\mathbf{x}_\mathbf{f} - \mathbf{x}(t_f)\right) = -\Delta\mathbf{x}_\mathbf{f}(\mathbf{p}) \tag{1}$$

$$\Delta\mathbf{p} = -\left[\frac{\partial \Delta\mathbf{x}_\mathbf{f}(\mathbf{p})}{\partial \mathbf{p}}\right]^{-1}\Delta\mathbf{x}_\mathbf{f}(\mathbf{p}) \tag{2}$$

If the Jacobian of the constraints is non-square (when the constraint vector and the parameter vector are not of equal length), a pseudo-inverse can be applied to produce the least squares or least norm solution.

Since the partial derivatives of the integral of the equations of motion cannot generally be found analytically, [Howard and Kelly 05] estimates must be found numerically. Forward (eq. 15) or central difference (eq. 16) linearizations of the forward equations of motion can be used to estimate these partial derivatives. The algorithm derives its vehicle independence from the numerical estimation of all of the partial derivatives.

$$\frac{\partial \Delta \mathbf{x}_{i,j}(\mathbf{p})}{\partial \mathbf{p}_k} = \frac{\Delta \mathbf{x}_{i,j}(\mathbf{p}_k + e, \mathbf{p}) - \Delta \mathbf{x}_{i,j}(\mathbf{p})}{e} \tag{3}$$

$$\frac{\partial \Delta \mathbf{x}_{i,j}(\mathbf{p})}{\partial \mathbf{p}_k} = \frac{\mathbf{q}_{i,j}(\mathbf{p}_k + e, \mathbf{p}) - \mathbf{q}_{i,j}(\mathbf{p}_k - e, \mathbf{p})}{2e} \tag{4}$$

Motion Prediction (Numerical Integration): The process of numerically integrating the differential equations that govern the motion of the vehicle is essentially one of simulation (Figure 1). Motion simulations are required to generate the trajectory corresponding to the parameterized controls and consequently the Jacobian and the Hessian of the Hamiltonian (because they are found by numerical linearizations of forward solutions). The process is broken into three distinct steps: control dynamics, modeling of the wheel-terrain interaction, and motion simulation.

The first step, modeling of the control dynamics, is very important for accurately simulating real-world systems. Real systems have motor acceleration and torque limits, latency, and joint limits that must be modeled in order to produce an accurate simulation of how the vehicle will respond to its commands. The control dynamics modeling portion of the motion prediction stage simply determines the vehicle response to the body-frame linear and angular velocity controls:

$$\dot{\mathbf{x}}(\mathbf{u}, t) = \mathbf{f}_{\textbf{control dynamics}}(\mathbf{x}, \mathbf{u}, t) \tag{5}$$

The control dynamics portion of the model determines the forces that the vehicle will exert on the environment, and therefore the resulting response forces from the environment. It is here where models of wheel slip, sliding, and the actions of external forces on the system are applied.

The last portion of the motion prediction is the actual motion simulation, which determines the change in vehicle pose ($\mathbf{r}|_{\textbf{world}}$, $\mathbf{o}|_{\textbf{world}}$) over a small time step (Δt):

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \dot{\mathbf{x}}(\mathbf{x}, \mathbf{u}, t)\Delta t \tag{6}$$

Since ground vehicles generally cannot control their roll, pitch, or elevation rates (they are functions of the interaction between the environment and the suspension), new estimates of these states are computed by the suspension model at the end of each motion prediction step.

Vehicle Model (Vehicle Simulation): From a coding perspective, the vehicle model portion of the trajectory generation algorithm is an instance of the abstract vehicle model class that determines how the robot will respond to the commanded body-frame linear and angular velocity controls, change its position and orientation, and interact with its environment (Figure 1). The vehicle model is generally broken down into four elements: a dynamics model, a wheel/terrain interaction model, a motion model, and a suspension model.

Vehicle Dynamics Models

Vehicle dynamics models simulate the response of the body-frame vehicle commands to the torques on and/or velocities of the wheels. By modeling the response to the parameterized controls, the solutions found are dynamically feasible.

A few elements of vehicle dynamics models are rate limits, joint limits, and latency. Rate limits represent constraints on how fast actuators can turn. Modeling these effects can account for drive wheels that move slower than requested or steering servos that lag behind their desired orientation. By contract, joint limits bound entire regions of control space that contain infeasible motions. An example of such infeasible motions is the impossible turn-in-place maneuver in automobiles. The front wheels can never reach the steer angles required for this motion. Accounting for latency in the system is essential for generating correct trajectories when the vehicle state can change dramatically over the scale of the latency.

These types of constraints can be handled in the trajectory generator by using the response to the input controls in the integration of the kinetic motion model instead of the controls themselves. Hard limits can be imposed on steering angles and models of actuator dynamics can simulate delays in the motor controller.

The imposition of such joint and rate limits in the vehicle model requires a method for computing wheel direction and velocities given some body-frame linear and angular velocity. For determining the mapping from body-frame linear and angular velocity to wheel velocities, suspension articulation rates can be temporarily neglected to simplify the computations. For such a rigid body, the velocity of any point on the

vehicle can be determined from the linear and angular velocity of the body frame. Specifically at the location of a wheel:

$$\mathbf{v}\big|_{\mathbf{wheel}} = \mathbf{v}\big|_{\mathbf{body}} + \boldsymbol{\omega}\big|_{\mathbf{body}} \times \mathbf{r}\big|_{\mathbf{wheel}} \tag{7}$$

In the linear and angular velocity vectors, only the controllable velocities (v_x,v_y,ω_z) need to be considered. The vector $\mathbf{r}|_{\mathbf{wheel}}$ is the displacement from the body frame to wheel contact point. The wheel speed and direction can then be determined from the computed wheel velocity vector.

Wheel/Terrain Interaction Models

By predicting the wheel/terrain interaction in the planning stage of the overall autonomy system, rather than accounting for it in the execution stage, more dynamically feasible vehicle motion can be generated. Traversing high wheel slip environments is a major challenge for current planetary robotic systems, for example and the present algorithm promises to help compensate for such slip to the degree that it can be predicted by the specialized perception algorithms that are appearing.

Vehicle Motion Models

The vehicle motion model maps body frame linear and angular velocities to world frame position and orientation rates. A body-frame coordinate system is defined with the positive x-axis pointing forward, the positive y-axis pointing to the right, and the positive z-axis pointing down. The mapping of linear velocities from the body-frame ($\mathbf{v}\big|_{\mathbf{body}}$) to world frame ($\mathbf{v}\big|_{\mathbf{world}}$) is completed by rotating the body-frame x, y, and z axes by the Euler angles roll (□), pitch (□), and yaw (□) respectively:

$$\dot{\mathbf{r}}\big|_{\mathbf{world}} = \mathbf{v}\big|_{\mathbf{world}} = \mathbf{ROT_z}(\psi)\mathbf{ROT_y}(\theta)\mathbf{ROT_x}(\phi)\mathbf{v}\big|_{\mathbf{body}} \tag{8}$$

$$\dot{\mathbf{r}}\big|_{\mathbf{world}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} = \begin{bmatrix} c\psi\, c\theta & c\psi\, s\theta s\phi - s\psi\, c\phi & c\psi\, s\theta c\phi + s\psi\, s\phi \\ s\psi\, c\theta & s\psi\, s\theta s\phi + c\psi\, c\phi & s\psi\, s\theta c\phi - c\psi\, s\phi \\ -s\theta & c\theta s\phi & c\theta c\phi \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{9}$$

This relationship can be inverted to determine the body-frame velocity vectors in terms of the global position rates:

$$\mathbf{v}\big|_{\mathbf{body}} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} c\psi\, c\theta & s\psi\, c\theta & -s\theta \\ c\psi\, s\theta s\phi - s\psi\, c\phi & s\psi\, s\theta s\phi + c\psi\, c\phi & c\theta s\phi \\ c\psi\, s\theta c\phi + s\psi\, s\phi & s\psi\, s\theta c\phi - c\psi\, s\phi & c\theta c\phi \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \tag{10}$$

The mapping between the Euler angle rates and the body-frame angular velocities ($_{\mathbf{body}}$) can also be found by transforming the individual Euler rotation rates from their intermediate frames to the robot-fixed frame:

$$\boldsymbol{\omega}\big|_{\mathbf{body}} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = \begin{bmatrix} \dot{\phi} \\ 0 \\ 0 \end{bmatrix} + \mathrm{rot}_x(\phi) \begin{bmatrix} 0 \\ \dot{\theta} \\ 0 \end{bmatrix} + \mathrm{rot}_x(\phi)\mathrm{rot}_y(\theta) \begin{bmatrix} 0 \\ 0 \\ \dot{\psi} \end{bmatrix} \tag{11}$$

$$\boldsymbol{\omega}\big|_{\mathbf{body}} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & s\theta \\ 0 & c\phi & -s\phi c\theta \\ 0 & s\phi & c\phi c\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \tag{12}$$

Just as with global position rates and body-frame linear velocities, this relationship can be inverted to determine the Euler rates in terms of the body-frame angular velocities:

$$\Psi\big|_{\mathbf{world}} = \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & t\theta s\phi & -t\theta c\phi \\ 0 & c\phi & s\phi \\ 0 & -\dfrac{s\phi}{c\theta} & \dfrac{c\phi}{c\theta} \end{bmatrix} \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \tag{13}$$

Equations (9) and (13) form the basis of the time-based velocity kinematics. The rate of change of global position and Euler angles can be determined given a set of controllable body-frame linear and angular velocities:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} c\psi\, c\theta & c\psi\, s\theta s\phi - s\psi\, c\phi & 0 \\ s\psi\, c\theta & s\psi\, s\theta s\phi + c\psi\, c\phi & 0 \\ 0 & 0 & \dfrac{c\phi}{c\theta} \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega_z \end{bmatrix} \tag{14}$$

Vehicle Suspension Models

Vehicle suspension modeling is the problem of properly determining the attitude and elevation of a vehicle given its current state. This is an embedded optimization problem because it is often highly underdetermined. We solve this problem by conceptually allowing the vehicle to float in altitude and attitude, and articulate in suspension, while minimizing the residual between the wheels and the terrain elevations under (or above) them.

REFERENCES

[1] T.M. Howard and A. Kelly. Trajectory Generation on Rough Terrain Considering Actuator Dynamics. In Proceedings of the *5th International Conference on Field and Service Robotics*, pp. 479-490, Port Douglas, Australia, July 2005. Berlin: Springer-Verlag.

[2] T.M. Howard and A. Kelly. Terrain-Adaptive Generation of Optimal Continuous Trajectories for Mobile Robots. In Proceedings of the *8th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, Munich, Germany, September 2005.

[3] T.M. Howard, R.A. Knepper, and A. Kelly. Constrained Optimization Path Following of Wheeled Robots in Natural Terrain. In Proceedings of the *10th International Symposium on Experimental Robotics*, pp. 343-352, Rio de Janeiro, Brazil, July 2006. Berlin: Springer-Verlag.

[4] T.M. Howard and A. Kelly. Trajectory and Spline Generation for All-Wheel Steering Mobile Robots. In Proceedings of the *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 4827-4832, Beijing, China, October 2006.

[5] T.M. Howard and A. Kelly. Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots. *International Journal of Robotics Research*. Vol. 26, No. 2, pp. 141-166. February 2007.

[6] A. Kelly and T.M. Howard. Terrain Aware Inversion of Predictive Models for Planetary Rovers. In Proceedings of the *NASA Science and Technology Conference*, Baltimore, MD, USA, April 2007.

[7] M. Pivtoraiko, T.M. Howard, I. Nesnas, and A. Kelly. Field Experiments in Rover Navigation via Model-Based Trajectory Generation and Nonholonomic Motion Planning in State Lattices. In Proceedings of the *9th International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, Los Angeles, CA, USA, February 2008.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Rough Terrain Trajectory Generation
DOCUMENT TYPE Trajectory Generator Detailed Design Document
AUTHOR Thomas M. Howard
REVISION DATE September 25th, 2006

DETAILED DESIGN LOCATION: The detailed design of the document is provided through the commented code and the corresponding Doxygen documentation.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Rough Terrain Trajectory Generation
DOCUMENT TYPE Trajectory Generator API Specification and User Guide Document
AUTHOR Thomas M. Howard
REVISION DATE September 25th, 2006

INTRODUCTION: This section details the trajectory generator API and details how the class is used.

REQUIRED FILES:

```
libTrajectoryGenerator.a        // the compiled trajectory generation library
TrajectoryGenerator.h           // the trajectory generation class header file
VehicleState.h                  // vehicle state definition class header file
VehicleModel.h                  // vehicle model abstract base class header file
VehicleControls.h               // vehicle controls class header file
Path.h                          // path class header file
HeuristicTable.h                // heuristic table class
GenericVehicleModel.h           // generic vehicle model class (inherited from
//    VehicleModel.h)
31x31x31x11LookupTable.txt      // 4-D heuristic lookup table (loaded at trajectory
//    generation initialization)
```

TRAJECTORY GENERATOR API SPECIFICATION

Functions:

void TrajectoryGenerator(VehicleModel *RobotModel): Class constructor which loads a specific vehicle model.
void SetVehicleModel(VehicleModel *RobotModel): Function which changes the vehicle model internal to the trajectory generator to the specified vehicle model.
bool SolveTrajectory(const VehicleState &initialState, const VehicleState &terminalState): Function which generates a trajectory between a pair of boundary states (initialState and terminalState).
VehicleControls GetVehicleControls(void): Function which returns the current values of the trajectory generators vehicle controls.
void SetVehicleControls(const VehicleControls &controls): Function which changes the controls internal to the trajectory generator to the specified controls.
Path GetPath(void): Function which returns the current Path generated by the trajectory generator.
void SetXPrecision(long double xPrecision): Function which sets the required accuracy of the terminal x position in meters. The default value is 0.01.
void SetYPrecision(long double yPrecision): Function which sets the required accuracy of the terminal y position in meters. The default value is 0.01.
void SetYawPrecision(long double yawPrecision): Function which sets the required accuracy of the terminal yaw orientation in radians. The default value is 0.01.
void SetKPrecision(long double kPrecision): Function which sets the required accuracy of the terminal curvature in radians/meters. The default value is 0.01.
void SetVPrecision(long double vPrecision): Function which sets the required accuracy of the terminal velocity in meters/second. The default value is 0.01.
void SetMaxIterations(int maxIterations): Function which sets the maximum number of iterations (corrections) before the routine returns false. The default value is 20.
void SetConvergenceRate(long double convergenceRate): Function which sets the rate of the forward step to take in the numerical optimization. The default value is 1.00 (100% of the forward step).
void SetTimeStep(cost long double &timeStep): Function which sets the dt (in seconds) for the forward solution simulation. The default value is 0.001.
void SetMaximumNumberSteps(const in &maximumNumberSteps): Function which sets the maximum number of steps to take in the forward solution simulation. The default value is 10000. Increasing this number may improve the accuracy of the forward simulation but increase runtime and vice versa.

Program Variables:

bool setHeuristic: Boolean which determines whether or not to load the heuristics. This should only be off in a table generation program. The default value is true.

bool exportPath: Boolean which determines whether or not to generate and store a sampled path at the end of a successfully solved trajectory. The default value is true.
Bool displayDebug: Boolean which determines whether or not to print out debugging printouts. The default value is true.

VEHICLE MODEL API: There are current three generic functions that specify a simple model of vehicle dynamics. Other function can be added by the user, but these three interface with the code in the forward solution section of trajectory generator

void ResetVehicleModelToInitialState(void): This function is called at the beginning of each forward solution to reset the vehicle model to its initial state.
void DynamicsResponse(VehicleState &currentState): This function passes in the current state and modifies it based on the response of a vehicle model (which the user adds). The inputted currentState contains the commanded linear velocity and curvature and the modified currentState is used which contains the response linear velocity and curvature. The generic vehicle model class assumes a unity response, where the response linear velocity and curvature are the commanded linear velocity and curvature.
void EvalSuspension(VehicleState &currentState): This function is used to determine the elevation, body roll, and pitch given some position and body yaw. This is again filled in by the user. The generic vehicle model class assumes that the elevation, body roll, and pitch are all equal to zero.

Example Program:

```
#include "TrajectoryGenerator.h"
#include "VehicleState.h"
#include "Path.h"
#include "VehicleControls.h"
#include "GenericVehicleModel.h"

using namespace TrajectoryGeneratorNamespace;

int main(){
        VehicleModel *RobotModel;
RobotModel = new GenericVehicleModel();
VehicleState initialState(0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0);
VehicleState terminalState(1.0,1.0,0.0,0.0,0.0,0.785,0.0,1.0,0.0);
TrajectoryGenerator TrajGen(RobotModel);
if(TrajGen.SolveTrajectory(initialState,terminalState)){
VehicleControls controls = TrajGen.GetVehicleControls();
Path path = TrajGen.GetPath();
}
return 0;
}
```

Example Program Description: Running the trajectory generator code relatively straightforward. First, include the necessary header files and link against the proper library (libTrajectoryGenerator.a). A VehicleModel defines the vehicle dynamics and suspension characteristics and is defined by the user using an instance of the abstract class VehicleModel. A GenericVehicleModel is included which assumes no suspension model (roll = pitch = z = 0.0) and perfect vehicle dynamics (vout = vin). Two VehicleState classes are instantiated which define the initial state of the vehicle (shown below as initialState) and the desired terminal state of the vehicle (shown below as terminalState). An instance of the TrajectoryGenerator class is generated using the class constructor which takes the current vehicle model as the argument. The standard class constructor, which takes no argument, uses the GenericVehicleModel instance of the VehicleModel class. Then to solve the trajectory generation problem, SolveTrajectory(initialState, terminalState) is called which returns true if the solution meets the required precision metrics. The controls and paths can be gathered from the trajectory generator by using the GetVehicleControls and GetPath functions respectively.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Rough Terrain Trajectory Generation
DOCUMENT TYPE Trajectory Generator Test Plan Document
AUTHOR Thomas M. Howard
REVISION DATE September 25th, 2006

INTRODUCTION: The trajectory generator shall be tested to determine if the functional, performance, interface, and reliability requirements have been met. This section will detail the test suite used to certify the algorithm and trace which requirements each test validates.

REGRESSION TESTS

- Test the algorithm with a large number of queries on flat terrain and compare solved trajectories to stored solutions. The controls parameterization and vehicle model shall be the default control parameterization (5th order polynomial in curvature, constant unity velocity, constant zero direction) and the default vehicle model (generic_vehicle_model) respectively.
    - (Functional Requirement R-1) Test the trajectory generator interface and report % convergence.
    - (Performance Requirement R-3) Test convergence and report % convergence.
    - (Interface Requirement R-8) If the trajectory generator properly generates trajectories between boundary states and returns the correct controls and vehicle path, then it is considered successful.
- Test the algorithm with a large number of queries on rough terrain (supplied) with the supplied Rocky 8 vehicle model with a large number of queries and compare solved trajectories to stored solutions. The controls parameterization shall be the default (5th order polynomial in curvature, constant unity velocity, constant zero direction). The tests would ideally be done on hardware comparable to Rocky 8 (for performance metrics).
    - (Functional Requirement R-1) Test the trajectory generator interface and report % convergence.
    - (Performance Requirement R-4) Test convergence on rough terrain and report % convergence. A % convergence above 99% on the supplied world is considered a success.
    - (Performance Requirement R-5) Measure and store average runtime of rough terrain trajectory generation tests. Average runtimes of less than 100 milliseconds shall be considered successful.
    - (Performance Requirement R-6) Measure and store memory footprint. A memory footprint of less than 20 megabytes is considered successful.
    - (Performance Requirement R-7) Measure and store initialization time. An initialization time of less than one second is considered a success.
    - (Interface Requirement R-9) If the trajectory generator successfully loads and uses the supplied Rocky 8 vehicle model, it is considered a success.
    - (Interface Requirement R-11) If the vehicle model can properly invert the suspension model to determine its attitude and elevation on the terrain, it is properly communicating with the terrain representation in the vehicle model and is considered a success.
- Test the algorithm with a large number of queries on rough terrain (supplied) with the supplied Rocky 8 vehicle model with specific dynamics enabled (delays, slip, etc…) and compare trajectories to stored solutions to measure algorithm performance and convergence.
    - (Interface Requirement R-12) If the vehicle model can properly predict vehicle dynamics as are described in the supplied vehicle model and compensate for these effects, it can properly communicate with the vehicle dynamics portion of the vehicle model and therefore is considered a successful test.

INTERFACE TESTS

- Test the algorithm by changing the vehicle controls parameterization multiple times and make sure that the mode has properly changed.
    - (Interface Requirement R-10) If the trajectory generator properly switches between different parameterization controls representations, it is considered a successful test.
- Test the algorithm by changing the vehicle model multiple times and make sure that the vehicle model has properly changed.

- (Interface Requirement R-9) If the trajectory generator properly switches between different vehicle models, it is considered a successful test.

ANYTIME ALGORITHM TESTS

- Test the algorithm with a small maximum algorithm time (50 milliseconds) to test if the program can properly supply a near-optimal trajectory in the "anytime" version of the algorithm.
  - (Functional Requirement R-2) Ensure that the algorithm can provide a near-optimal solution based on a limited runtime, effectively limiting the maximum number of optimization steps. If the trajectory generator can properly provide a reasonable trajectory with state constraint error less than the initial state constraint error, it is considered a successful test.
  - (Reliability Requirment R-17) If the trajectory generator properly reports that the maximum time limit has been exceeded, then it is considered a successful test.

RELIABILITY TESTS

- Provide the trajectory generator with unreasonable (NaN, inf, etc…) state constraints to tests for proper error handling.
  - (Reliability Requirement R-13) If the trajectory generator exits and reports an implausible state constraint then it is considered a successful test.
- Provide the trajectory generator with an improper control parameterization to test for proper error handling.
  - (Reliability Requirement R-14) If the trajectory generator exits and reports an implausible control parameterization then it is considered a successful test.
- Provide the trajectory generator with an improper vehicle model (bad suspension model, control dynamics model, etc…) to test for proper error handling.
  - (Reliability Reqruiement R-15) If the trajectory generator exits and reports an implausible vehicle model then it is considered a successful test.
- Provide the trajectory generator with an infeasible trajectory generation problem to detect non-convergence or divergence of the algorithm (exceeding maximum # of iterations)
  - (Reliaiblity Requriement R-16) If the trajectory generator exits and reports an divergence if the Jacobian becomes singular or if the maximum number of iterations has been exceeded (non-convergence) then it is considered a successful test.

PROJECT Very Rough Terrain Trajectory Generation and Motion Planning for Planetary Rovers
TASK Rough Terrain Trajectory Generation
DOCUMENT TYPE Trajectory Generator Test Results Document
AUTHOR Thomas M. Howard
REVISION DATE August 31st, 2007

INTRODUCTION: This section states that the rough terrain trajectory generator has successfully passed the provided test program and tracks all of the software requirements. The output from the successfully passed tests can be found in the Surface_Parameterized_Trajectory_Generator module and should be used to perform regression test to modifications of the algorithm.

Test ID	Description	Result	Test Date
R1	(Functional Requirement R-1) Test the trajectory generator interface and report % convergence.	PASSED	AUGUST 2007
R2	(Performance Requirement R-3) Test convergence and report % convergence.	PASSED	AUGUST 2007
R3	(Interface Requirement R-8) If the trajectory generator properly generates trajectories between boundary states and returns the correct controls and vehicle path, then it is considered successful.	PASSED	AUGUST 2007
R4	(Functional Requirement R-1) Test the trajectory generator interface and report % convergence.	PASSED	AUGUST 2007
R5	(Performance Requirement R-4) Test convergence on rough terrain and report % convergence. A % convergence above 99% on the supplied world is considered a success.	PASSED	AUGUST 2007
R6	(Performance Requirement R-5) Measure and store average runtime of rough terrain trajectory generation tests. Average runtimes of less than 100 milliseconds shall be considered successful.	PASSED	AUGUST 2007
R7	(Performance Requirement R-6) Measure and store memory footprint. A memory footprint of less than 20 megabytes is considered successful.	PASSED	AUGUST 2007
R8	(Performance Requirement R-7) Measure and store initialization time. An initialization time of less than one second is considered a success.	PASSED	AUGUST 2007
R9	(Interface Requirement R-9) If the trajectory generator successfully loads and uses the supplied Rocky 8 vehicle model, it is considered a success.	PASSED	AUGUST 2007
R10	(Interface Requirement R-11) If the vehicle model can properly invert the suspension model to determine its attitude and elevation on the terrain, it is properly communicating with the terrain representation in the vehicle model and is considered a success.	PASSED	AUGUST 2007
R11	(Interface Requirement R-12) If the vehicle model can properly predict vehicle dynamics as are described in the supplied vehicle model and compensate for these effects, it can properly communicate with the vehicle dynamics portion of the vehicle model and therefore is considered a successful test.	PASSED	AUGUST 2007
R12	(Interface Requirement R-10) If the trajectory generator properly switches between different parameterization controls representations, it is considered a successful test.	PASSED	AUGUST 2007
R13	(Interface Requirement R-9) If the trajectory generator properly switches between different vehicle models, it is considered a successful test.	PASSED	AUGUST 2007
R14	(Functional Requirement R-2) Ensure that the algorithm can provide a near-	PASSED	AUGUST 2007

	optimal solution based on a limited runtime, effectively limiting the maximum number of optimization steps. If the trajectory generator can properly provide a reasonable trajectory with state constraint error less than the initial state constraint error, it is considered a successful test.		
R15	(Reliability Requirment R-17) If the trajectory generator properly reports that the maximum time limit has been exceeded, then it is considered a successful test.	PASSED	AUGUST 2007
R16	(Reliability Requirement R-13) If the trajectory generator exits and reports an implausible state constraint then it is considered a successful test.	PASSED	AUGUST 2007
R17	(Reliability Requirement R-14) If the trajectory generator exits and reports an implausible control parameterization then it is considered a successful test.	PASSED	AUGUST 2007
R18	(Reliability Reqruiement R-15) If the trajectory generator exits and reports an implausible vehicle model then it is considered a successful test.	PASSED	AUGUST 2007
R19	(Reliaiblity Requriement R-16) If the trajectory generator exits and reports an divergence if the Jacobian becomes singular or if the maximum number of iterations has been exceeded (non-convergence) then it is considered a successful test.	PASSED	AUGUST 2007