UT/JSC Trick Modeling Initiative 2010

Iterative Targeting Algorithm in the Monte Carlo Framework

Faculty Sponsor:
Dr. Belinda Marchand

Project Manager:
Kyle Brill

Team Leaders:
Zach Tschirhart, Field Manar

Group Members:
Travis Sanders, Chun-Yi Wu, James Dearman, Szu-Chun Hung, Victor Rodriguez,
Michael Tarng, Chris Cutlip, Harsh Shah

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

The present study focuses on the implementation of an iterative targeting algorithm within a Trick simulation environment. A team of undergraduate research students at The University of Texas at Austin accomplished the investigation and software implementation. The simulation developed seeks to identify the impulsive maneuver required by a spacecraft to transfer between two arbitrary points in Earth orbit. The initial state and the terminal position vector are inputs to the Trick simulation. Then, Trick's built-in Monte Carlo master/slave framework is leveraged to converge on the impulsive maneuver required for the spacecraft to initiate the transfer. The spacecraft trajectory modeling is accomplished through Trick with simple, two-body, point mass equations of motion.

The targeting algorithm relies on the availability of the state transition matrix (STM). To facilitate a generalized force model and future JSC Engineering Orbital Dynamics (JEOD) implementation, the STM is constructed with a finite differencing algorithm. Thus, the Trick simulation developed in this report is divided into three routines: the physical simulation, the finite differencing process, and the targeting process.

The physical simulation consists of a set of differential equations integrated with Trick. The finite differencing process calculates the STM. Finally, the targeting process employs the STM to identify the necessary changes in the initial state to achieve the desired end state. This work builds on the results of a previous study of optimization in Trick and the Cannonball tutorial provided with the Trick Documentation.

The body of this final report presents the objectives of this project, the mathematical grounding of the algorithm, and discussion of the interaction between the various Trick source and model files. Finally, the results are illustrated, along with some of the major obstacles encountered in an attempt to integrate the targeter with JEOD. Suggestions are made for future study of the interaction between the Monte Carlo framework and JEOD, and the implementation of more advanced targeters in Trick.

1.0 INTRODUCTION

1.1 Motivation

The purpose of this study is to explore the implementation of an iterative targeting algorithm using NASA’s simulation toolkit, Trick. This simple algorithm provides a springboard for models of increased complexity, scope, and practical value. For further discussion on studies of targeting for NASA's Orion project, and more advanced algorithms, see Marchand [1].

1.2 Goals

The two goals of this project are to implement, in Trick, a non-real time iterative targeter that employs a Level 1 differential corrections process and to integrate the JSC Engineering Orbital Dynamics (JEOD) package into the underlying trajectory simulation. Previous work explored the implementation of an iterative Trick environment [2]. The present study expands the capabilities and utility of this earlier environment.

Incorporation of JEOD will allow for a generalized algorithm of arbitrary dynamical complexity. Initial work with JEOD focuses on learning C++, understanding the tool from an operator’s perspective, running the Tutorial simulations, and becoming familiar with the package as this is the first year of its use at UT.

2.0 BACKGROUND

2.1 Project Breakdown

The design of the Trick targeting algorithm entails three primary components: the targeting process itself, the finite differencing process, and the trajectory modeling (either with a user provided model or JEOD). The mathematical notions involved in targeting are outside the scope of undergraduate studies. The same can be said about the notion of finite differencing for the construction of numerical derivatives. Furthermore, working with JEOD required a certain level of proficiency in C++. Accomplishing these three tasks required much self-education on all three subjects. The following sections are devoted to summarizing how each of these three tasks were studied and accomplished individually.

2.2 Targeting

In general, targeting is the process of iteratively changing control parameters in order to achieve some pre-specified goal. The control parameters, in this case, are the three components of an impulsive change in velocity executed at the beginning of a satellite's Earth orbit. The pre-specified goal, in this case, is an end-state constraint on position. The targeting algorithm is responsible for determining the impulsive maneuver required at the start of the trajectory in order for the spacecraft to reach its intended destination. This section outlines the use of the Level 1 targeter presented in [1].

Let the inertial position and velocity vectors be defined as $\boldsymbol{R}(t) = [x(t)\ y(t)\ z(t)]^T\ km$ and $\boldsymbol{V}(t) = [\dot{x}(t)\ \dot{y}(t)\ \dot{z}(t)]^T\ km/s,$ respectively. The state vector, then, is defined as $\boldsymbol{X}(t) = [\boldsymbol{R}(t)\ \boldsymbol{V}(t)]^T$, so that the state of the satellite at time $t = 0$ seconds is given by the following:

$$\boldsymbol{X}(t_o) = \begin{bmatrix} \boldsymbol{R}(t_o) \\ \boldsymbol{V}(t_o) \end{bmatrix}. \tag{1}$$

From the definition of the satellite state, the nonlinear differential equations that govern the propagation of $\boldsymbol{X}(t)$ forward in time can be represented by

$$\dot{\boldsymbol{X}}(t) = \boldsymbol{f}[\boldsymbol{X}(t)]. \tag{2}$$

Any reference trajectory of interest, $\boldsymbol{X}^*(t)$, must satisfy equation (2) such that

$$\dot{\boldsymbol{X}}^*(t) = \boldsymbol{f}[\boldsymbol{X}^*(t)]. \tag{3}$$

The reference solution employed in this study is defined as the "current" solution. Thus, the reference solution changes after each iteration of the algorithm, as each update is processed. The nonlinear state of the satellite may then be defined relative to the reference path $\boldsymbol{X}^*(t)$ by

$$\boldsymbol{X}(t) = \boldsymbol{X}^*(t) + \delta\boldsymbol{X}(t), \tag{4}$$

where $\delta\boldsymbol{X}(t)$ is the state deviation measured relative to the reference path, $\boldsymbol{X}^*(t)$. Linearizing Equation (2) about the reference trajectory leads to

$$\delta\dot{\boldsymbol{X}}(t) = A(t)\delta\boldsymbol{X}(t), \tag{5}$$

where $A(t) = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{X}}$, the Jacobian, is evaluated along $\boldsymbol{X}^*(t)$. Equation (5) admits a solution of the form

$$\delta\boldsymbol{X}(t) = \Phi(t, t_0)\delta\boldsymbol{X}(t_0). \tag{6}$$

Here, $\Phi(t, t_0)$ is termed the state transition matrix (STM). The STM originates from the solution to a matrix differential equation [3]:

$$\dot{\Phi}(t, t_0) = A(t)\Phi(t, t_0). \tag{7}$$

Subject to an initial condition, $\Phi(t_0, t_0) = I$, where I is an identity matrix of the appropriate dimensions.

The nonlinear relationship between the constraints and the control parameters in a targeting process is formulated as $\boldsymbol{c} = \boldsymbol{g}(\boldsymbol{p})$, where $\boldsymbol{c}$ denotes a vector of constraints and $\boldsymbol{p}$ is the vector of control parameters. Along the reference trajectory, then, $\boldsymbol{c}^* = \boldsymbol{g}(\boldsymbol{p}^*)$. Thus, a linearization about the reference trajectory suggests that $\delta\boldsymbol{c} = M\delta\boldsymbol{p}$, where M denotes some time varying matrix. If the number of constraints is equal to the number of control parameters, then this equation admits only one solution. If, however, the number of constraints is less than the number of control parameters, it admits an infinite number of solutions. In this case, one commonly employed solution is known as the minimum norm solution. That is, the solution that minimizes $\delta\boldsymbol{p}$:

$$\delta\boldsymbol{p} = M^T(\mathrm{MM^T})^{-1}\delta\boldsymbol{c}. \tag{8}$$

The state relationship matrix, M, in this case, depends on the state transition matrix determined either with (7), or the finite differencing formulation in section 2.3. For a derivation of equation (8), see Bate [3] or Corless [4]. In equation (8), $\delta\boldsymbol{p}$ represents a small change to the vector of control parameters, which will subsequently inflict a small change on the value of the constraints, $\delta\boldsymbol{c}$. In the initial algorithm considered, $\delta\boldsymbol{p} = \delta\boldsymbol{v_o}$, where $\delta\boldsymbol{v_o}$ is the impulsive burn at time t_o. Also, $\delta\boldsymbol{c}$ represents the error in the terminal position vector:

$$\delta\boldsymbol{c} = \boldsymbol{r_d} - \boldsymbol{R}(\boldsymbol{t_f}), \tag{9}$$

where $\boldsymbol{r_d}$ is the desired final position and $\boldsymbol{R}(t_f)$ is the actual terminal position of the spacecraft. Figure 1 illustrates an example of these relationships similar to the enclosed implementation.

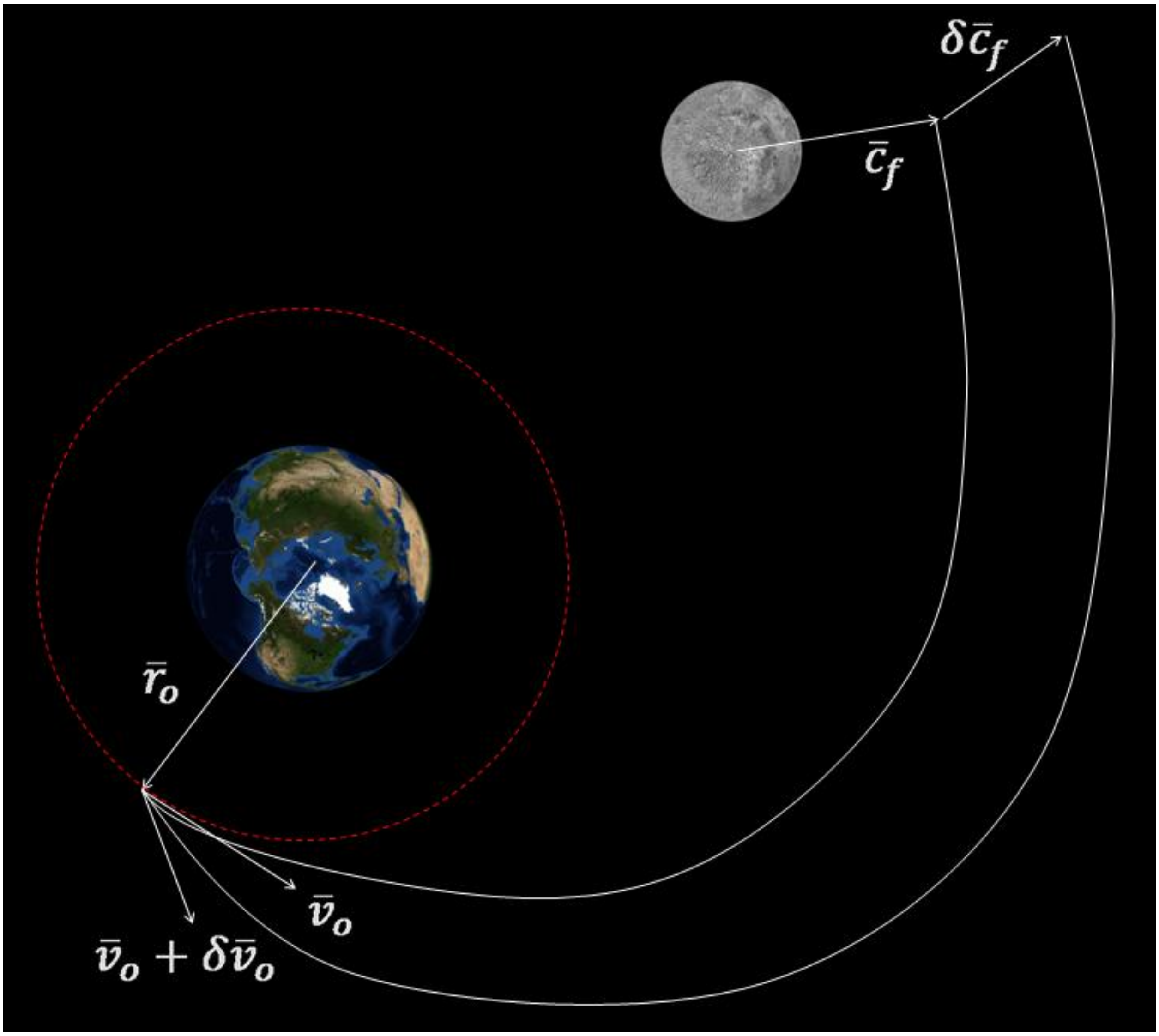


Figure 1: Visual representation of the targeting algorithm

Since the initial targeter considered is sufficiently simple, and B is a square matrix, it turns out that

$$M^T(\mathrm{MM^T})^{-1} = B(t_f, t_0)^{-1}, \tag{10}$$

where B is the top right 3x3 matrix of $\Phi(t_f, t_0)$:

$$\Phi(t_f, t_0) = \begin{bmatrix} A(t_f, t_o) & B(t_f, t_o) \\ C(t_f, t_o) & D(t_f, t_o) \end{bmatrix}. \tag{11}$$

Thus, the final targeting equation is

$$\delta \boldsymbol{v_o} = B^{-1}\delta \boldsymbol{r}. \tag{12}$$

2.3 Finite Differencing

For the simple case of a satellite orbiting an inertially fixed Earth in the two-body point mass problem, the Jacobian is easily constructed. This allows the integration of the STM with equation (7). However, as the dynamical model becomes more complex, these partials are both difficult to evaluate and cumbersome to determine. Furthermore, integrating JEOD into the targeting process does not lend itself to the use of analytical partial derivatives. Constructing the STM using finite differencing (FD) is an appealing option that simplifies the use of JEOD, or any other "black box" physical model, in the targeting process. Finite differencing allows the targeting algorithm to compute the STM without direct knowledge of the force model in equation (2). The method employed here takes advantage of the fact that the STM is essentially a linear sensitivity matrix. The process works in the following manner. First, specify a small perturbation for each state:

$$\boldsymbol{\delta_1} = \begin{bmatrix} \delta_1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \boldsymbol{\delta_2} = \begin{bmatrix} 0 \\ \delta_2 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \dots \boldsymbol{\delta_6} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \delta_6 \end{bmatrix}. \tag{13}$$

The impact of each of these perturbations on the terminal state is considered individually. That is, the initial state is perturbed first by $\boldsymbol{\delta_1}$ and the impact on the terminal state is measured. Then, the process is repeated for $\boldsymbol{\delta_2}$ through $\boldsymbol{\delta_6}$. Notice that each of these $\boldsymbol{\delta_i}$'s considers only a small perturbation in one of the six states. The structure of each perturbation vector is important in calculating the STM. The perturbed initial vectors look like this:

$$\boldsymbol{X_1}(t_o) = \boldsymbol{X}^*(t_o) + \boldsymbol{\delta_1}, \dots \boldsymbol{X_6}(t_o) = \boldsymbol{X}^*(t_o) + \boldsymbol{\delta_6}. \tag{14}$$

Next, a 42 element state vector is constructed for numerical integration. This vector consists of the reference trajectory, followed by the six perturbed trajectories,

$$\tilde{\boldsymbol{X}}(t) = \begin{bmatrix} \boldsymbol{X}^*(t) \\ \boldsymbol{X_1}(t) \\ \boldsymbol{X_2}(t) \\ \vdots \\ \boldsymbol{X_6}(t) \end{bmatrix}, \tilde{X}_i = X_{j,k} \tag{15}$$

where j ranges from zero to six for the seven trajectory vectors and k indexes the elements in the j vector. Once the integration is performed, the final state associated with each perturbed initial state vector is available. Thus, the STM is approximated numerically as follows:

$$\Phi(t_f, t_0) \approx \frac{\partial \boldsymbol{X}(t_f)}{\partial \boldsymbol{X}(t_0)} = \begin{bmatrix} \frac{\tilde{X}_{1,1}(t_f) - X_1^*(t_f)}{\delta_1} & \cdots & \frac{\tilde{X}_{6,1}(t_f) - X_1^*(t_f)}{\delta_6} \\ \vdots & \ddots & \vdots \\ \frac{\tilde{X}_{1,6}(t_f) - X_6^*(t_f)}{\delta_1} & \cdots & \frac{\tilde{X}_{6,6}(t_f) - X_6^*(t_f)}{\delta_6} \end{bmatrix} \tag{16}$$

Equation (16) can then be applied in conjunction with (11) and (12) to target the desired constraint.

The finite differencing method employed relies on the assumption that the deviations in the terminal state are in the “linear” range. Thus, the parameter δ_j must be carefully selected to preserve this assumption. It is also true, however, that they should not be set too small because this can introduce convergence difficulties. The accuracy of the presented STM approximation will, of course, depend on the quality of the δ_j's selected. The quality of this choice can, in a simplified model, be validated against the numerically integrated state transition matrix.

2.4 Trajectory Simulation

The equations of motion used to propagate the trajectory for the initial, user defined, physical model extend from the `Ball++` code included with Trick [5]. They are in three dimensions and assume two point mass bodies with the Earth inertially fixed. A flow chart of the code is depicted in Figure 2. Each arrow denotes a step down to the next directory level of the code.

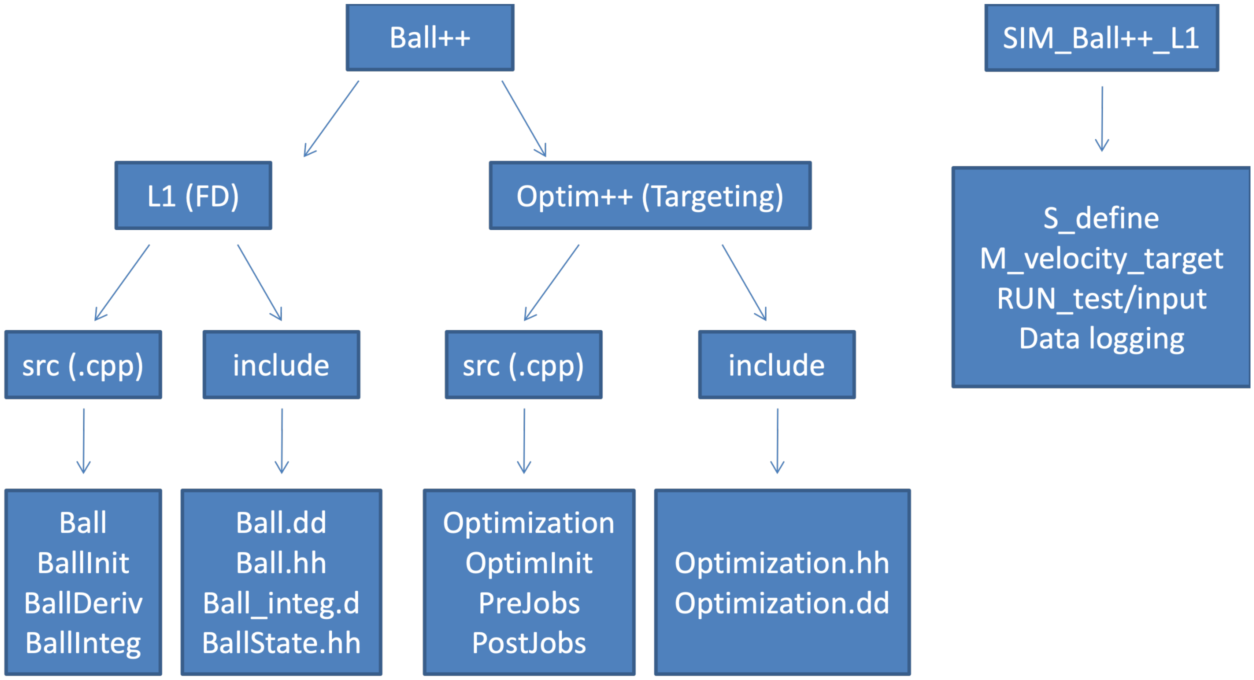


Figure 2: Code Structure for User Provided Equations of Motion

The Finite Differencing will take place in the `L1` area of the code, while the Targeting resides in `Optim++`. The Monte Carlo job which defines the interface between the two is in the `SIM_Ball++_L1` directory. Specifically, the framework is declared in the `S_define` file. Next, a flow chart of the Monte Carlo optimization framework is shown in Figure 3.

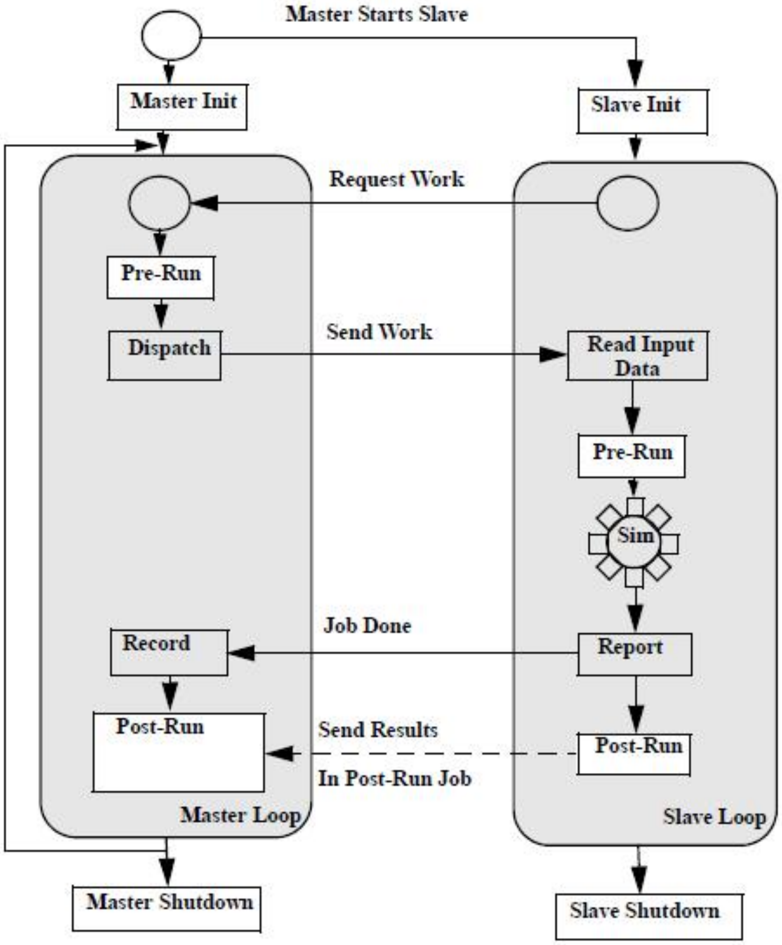


Figure 3: Master Slave Framework for Optimization [5]

The goal is to wrap the targeting algorithm in the master loop around the slave which contains either the `Ball++` equations of motion or, for the second part of the project, the desired JEOD model. Figure 4 shows the flow diagram of the JEOD code when integrating with the targeter.

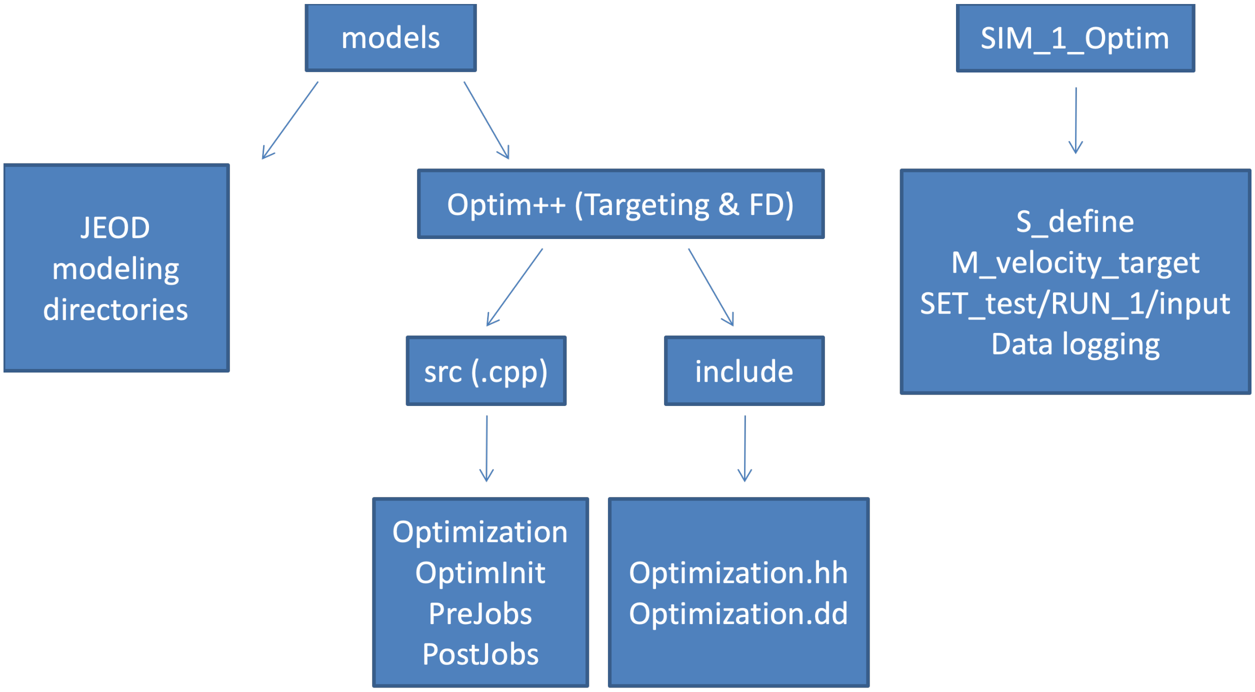


Figure 4: Code Structure for JEOD Integration

Both the specifics of this framework and how the pieces operate in tandem will be further developed in the following sections. For more information on the JEOD modeling directories, see Jackson [6]. All of the code discussed in this section can be found in the Appendices.

3.0 IMPLEMENTATION

3.1 Finite Differencing

First, as verification of the finite differencing process, the method for calculating $\Phi(t_f, t_0)$ outlined in section 2.3 was implemented in MATLAB. It is assumed that the satellite is initially on a circular orbit about the central body. The finite differencing computation agreed with the matrix differential equation outlined in section 2.2 which was implemented by the Targeting group.

This paved the way for the implementation of Finite Differencing in Trick. In the model source directory, a method for calculating the STM was added. Also, the default ball integration file was updated to allocate 42 integration variables and use the fourth order Runge-Kutta method. Consequently, this required updating the satellite state objects to deal with multiple trajectories instead of just one.

In the initialization phase of the code, the initial conditions and perturbations are stored from the appropriate default data file. The three-dimensional equations of motion are coded into the derivative class, while interfacing with the Trick Runge-Kutta 4 integration method is taken care of in the integration class.

3.2 Targeting

The targeting algorithm defined by equation (12), and the framework to support it, was coded into MATLAB. This allowed for a quick check of the algorithms functionality as well as providing a proving ground to learn the basic elements of a targeting algorithm.

The targeting algorithm was then transitioned into the optimization framework. Previously, the pre-run jobs in the master loop had incremented a ball "jet" firing time with no decision-making capacity except to terminate the program when a firing time reached some upper limit.

To implement the targeting algorithm, the bulk of the calculations are placed in the post jobs section of the master function found in `OptimPostJobs.cpp`. After the slave has returned, the STM is calculated with a call to the finite differencing function. This left the pre-master the simple task of checking stopping conditions, updating the initial conditions to match the new initial velocity, and passing that information along to the slave simulation.

The targeting team then looked into other, slightly more advanced, targeting algorithms based on equation (8). The first was a time free position-targeting algorithm. The second was a time free altitude-targeting algorithm. The final was a two impulse position-targeting algorithm. Of these, the first two were successful. The third, however, had issues with convergence and is still under investigation.

While a time free MATLAB targeter was successfully implemented, some difficulties were encountered when attempting to implement the same algorithm in Trick. After some investigation, it was determined that the stop time from the `RUN_test/input` file is stored in `sys.exec.work.terminate_time` field through the input processor. However, when the termination time was modified in the master object, the integration time for the slave object did not change. As this investigation occurred late in the project, and it is still unresolved, this task was left as an item for future work.

3.3 JEOD

The JEOD sample simulations were successfully executed and used as training material during this investigation. A comparison of the plots generated for a Low Earth Orbit by the simple gravity JEOD model in Tutorial SIM_1 vs. the user defined EOM is shown in Figure 5.

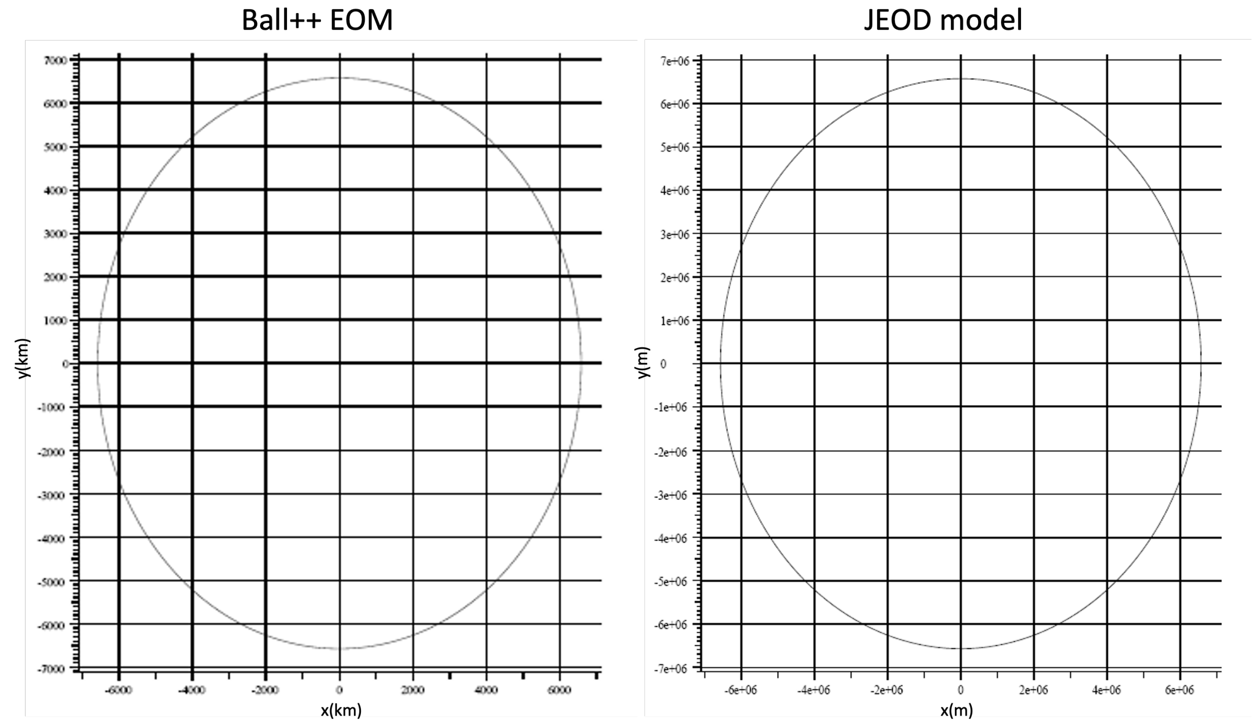


Figure 5: LEO comparison with JEOD

It is seen that both produce the same plot, with JEOD using meters whereas the user coded simulation used km.

4.0 INTEGRATION

4.1 FD to Targeting

The most important part of the integration is passing the slave object to the master object in the simulation. This allows for the modification of the ball input state in the pre-master and gives the optimizer access to the STM calculation method.

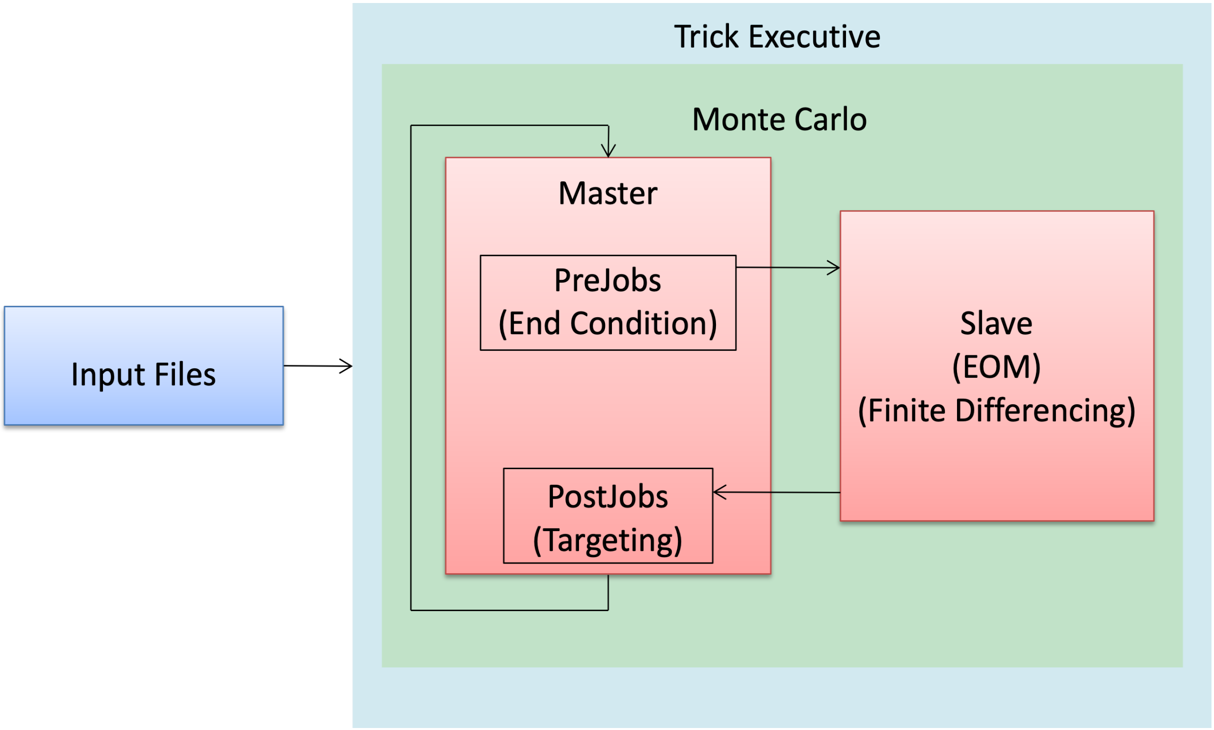


Figure 6: Depiction of the targeter for a user defined model

In Figure 6, the pre-master passes the 42 element vector to the slave for integration and receives the integrated final state in the post master.

4.2 Targeting to JEOD

There are two main components associated with the integration effort with JEOD. The first is modifying the existing Tutorial's `S_define` file to include Monte Carlo framework. The second is dealing with the JEOD as a "black box" physical model to be used without modification.

Adapting the simulation definition to function with the targeter proved relatively easy, as the structure of the targeting object could be copied from the user defined case. Since the targeter needs to modify the satellite state through each iteration, it was determined that `sv_dyn` should be passed to the master since this contains the JEOD vehicle state information.

To keep the targeter separate from the model, the STM calculator was moved from the slave to the master. JEOD is not designed to propagate a 42 element vector. Thus, a modified method for calculating the STM had to be developed. The proposed method still relies on the creation of a 42 element state vector; however, it takes seven calls to the slave instead of one to build up the necessary perturbation information. This is shown in the following graphic.

```
switch (counter%7)

case 0:
     --Apply delta v to initial state in pre master
     --Store reference trajectory final state in post master
     --counter++
case 1:
     --Apply perturbation to the first element of the state vector in the pre master
     --Store the output state for the perturbed trajectory in the post master
     --counter++
                           .
                           .
                           .
case 6:
     --Apply perturbation to the sixth element of the state vector in the pre master
     --Store the output state, calculate the STM and apply targeting formula for a new delta v
     --counter++
```

Ideally, no JEOD code needs to change to interface with the targeting algorithm. Moreover, if each simulation is using the same satellite object name, the optimization loop could be copied into each SIM to produce a targeter since they all draw from the same file structure. Figure 7 shows the independence of the targeting algorithm and the JEOD simulation.

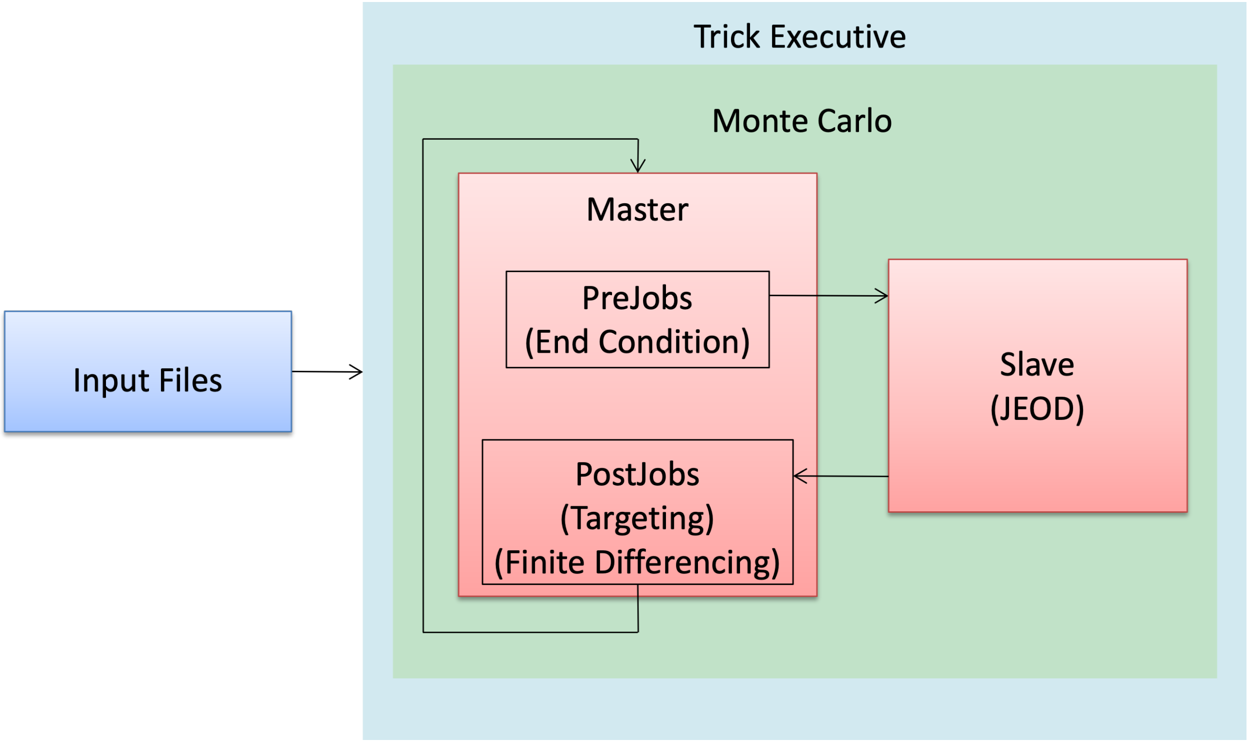


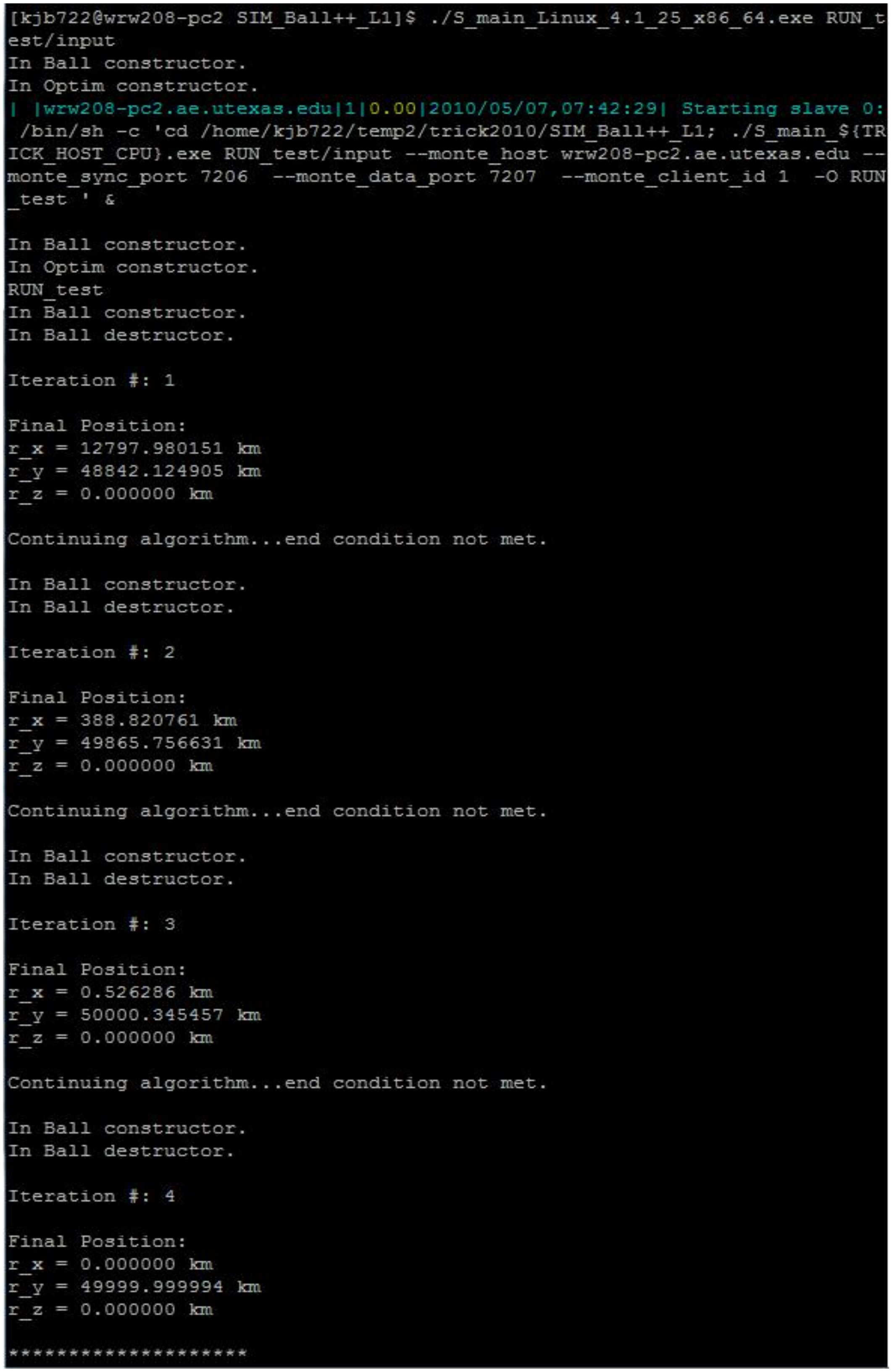
Figure 7: Depiction of the targeter when using JEOD

The information passed to the slave from the pre-master is the initial conditions for the desired trajectory integration. The post master receives the integrated final state from the slave.

5.0 RESULTS

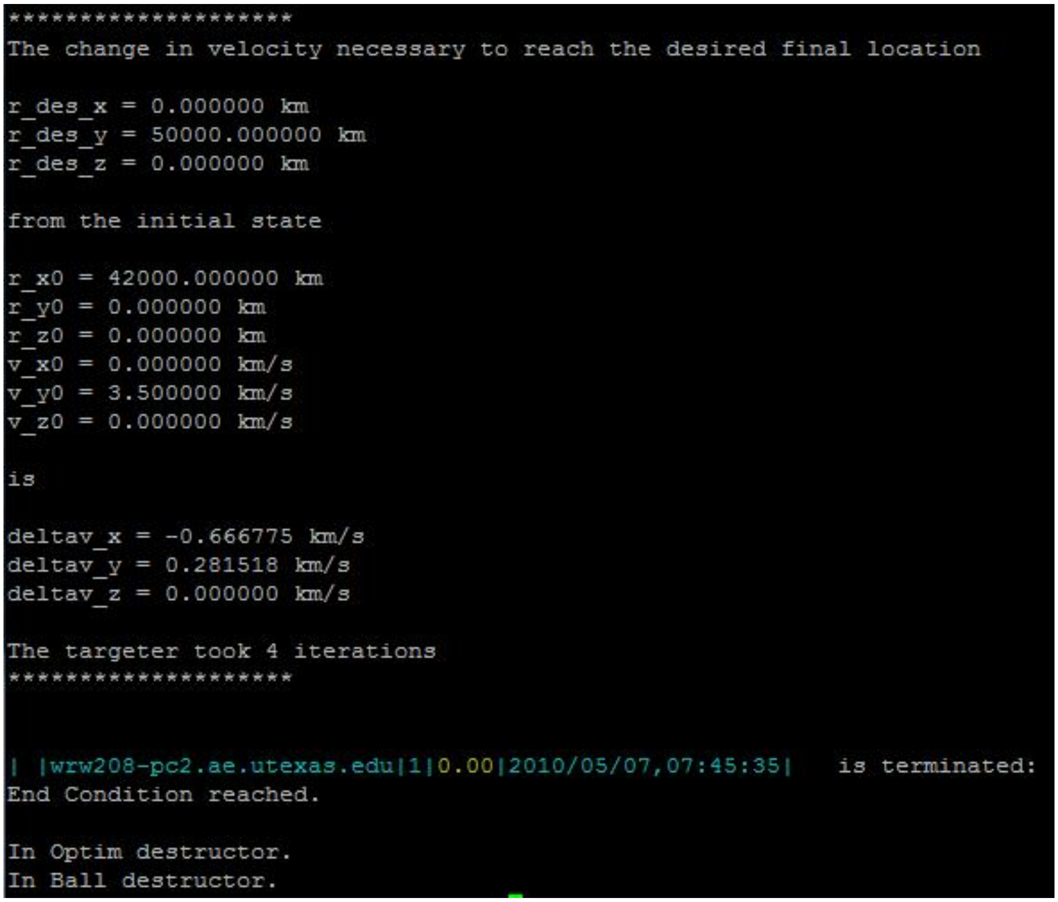
5.1 Targeting

To begin, we verified the Trick targeter with the MATLAB targeter. For an initial state of $\boldsymbol{R}(t_o)^T = [0, 42000, 0](km)$, $\boldsymbol{V}(t_o)^T = [0, 3.5, 0](km/s)$, both targeters computed $\Delta\boldsymbol{V}^T = [-0.667, 0.282, 0](km/s)$ to reach $\boldsymbol{R}\left(t_f\right)^T = [50000, 0, 0](km)$. We also checked the intermediate targeting curves of both programs and found that they were in agreement. The output from a successful Trick targeting simulation is displayed in Figures 8 and 9.



```
[kjb722@wrw208-pc2 SIM_Ball++_L1]$ ./S_main_Linux_4.1_25_x86_64.exe RUN_t
est/input
In Ball constructor.
In Optim constructor.
| |wrw208-pc2.ae.utexas.edu|1|0.00|2010/05/07,07:42:29| Starting slave 0:
 /bin/sh -c 'cd /home/kjb722/temp2/trick2010/SIM_Ball++_L1; ./S_main_${TR
ICK_HOST_CPU}.exe RUN_test/input --monte_host wrw208-pc2.ae.utexas.edu --
monte_sync_port 7206  --monte_data_port 7207  --monte_client_id 1  -O RUN
_test ' &

In Ball constructor.
In Optim constructor.
RUN_test
In Ball constructor.
In Ball destructor.

Iteration #: 1

Final Position:
r_x = 12797.980151 km
r_y = 48842.124905 km
r_z = 0.000000 km

Continuing algorithm...end condition not met.

In Ball constructor.
In Ball destructor.

Iteration #: 2

Final Position:
r_x = 388.820761 km
r_y = 49865.756631 km
r_z = 0.000000 km

Continuing algorithm...end condition not met.

In Ball constructor.
In Ball destructor.

Iteration #: 3

Final Position:
r_x = 0.526286 km
r_y = 50000.345457 km
r_z = 0.000000 km

Continuing algorithm...end condition not met.

In Ball constructor.
In Ball destructor.

Iteration #: 4

Final Position:
r_x = 0.000000 km
r_y = 49999.999994 km
r_z = 0.000000 km

********************
```

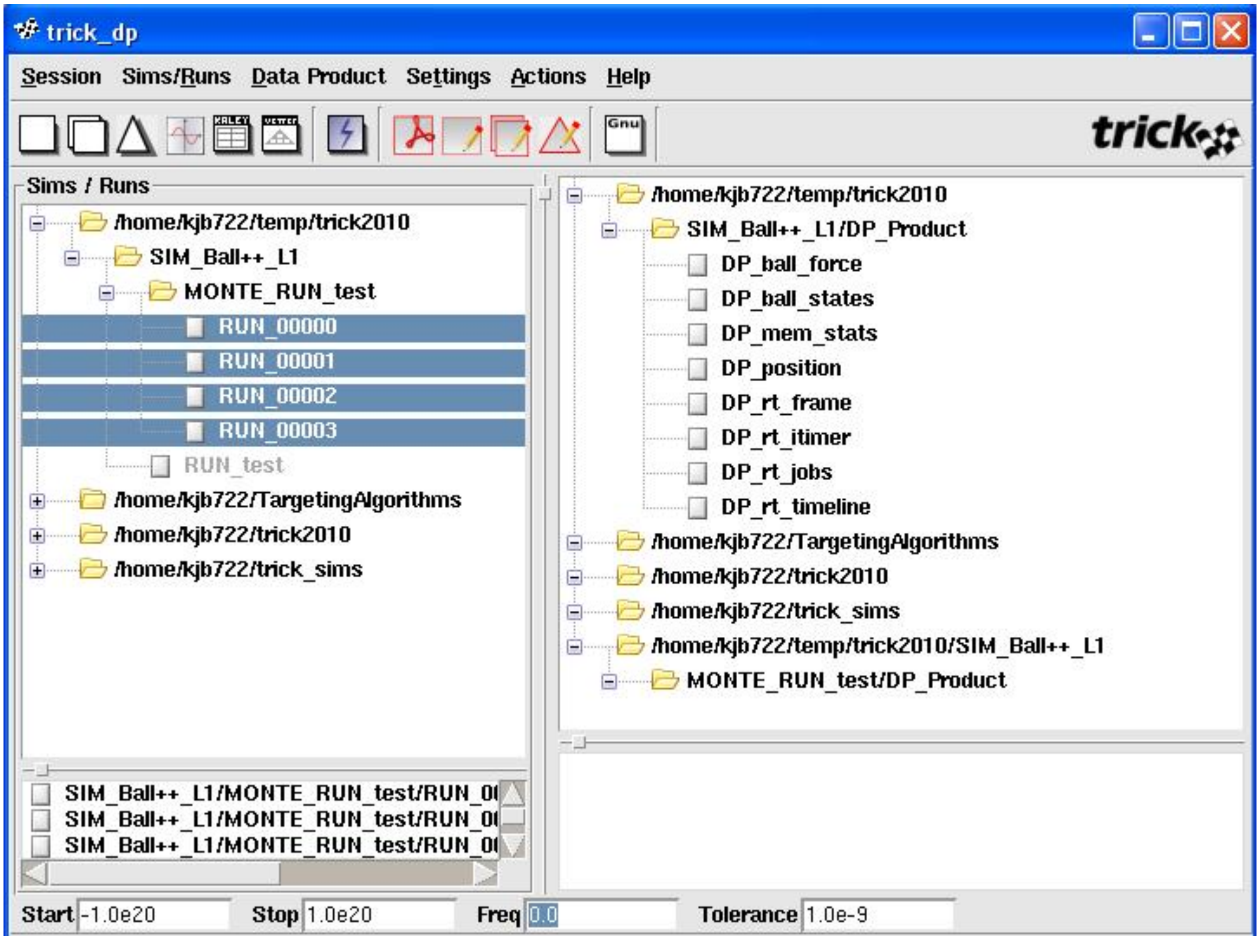
Figure 8: The first part of the output for a successful simulation run



```
********************
The change in velocity necessary to reach the desired final location

r_des_x = 0.000000 km
r_des_y = 50000.000000 km
r_des_z = 0.000000 km

from the initial state

r_x0 = 42000.000000 km
r_y0 = 0.000000 km
r_z0 = 0.000000 km
v_x0 = 0.000000 km/s
v_y0 = 3.500000 km/s
v_z0 = 0.000000 km/s

is

deltav_x = -0.666775 km/s
deltav_y = 0.281518 km/s
deltav_z = 0.000000 km/s

The targeter took 4 iterations
********************


| |wrw208-pc2.ae.utexas.edu|1|0.00|2010/05/07,07:45:35|   is terminated:
End Condition reached.

In Optim destructor.
In Ball destructor.
```

Figure 9: The change in velocity computed by the targeting algorithm

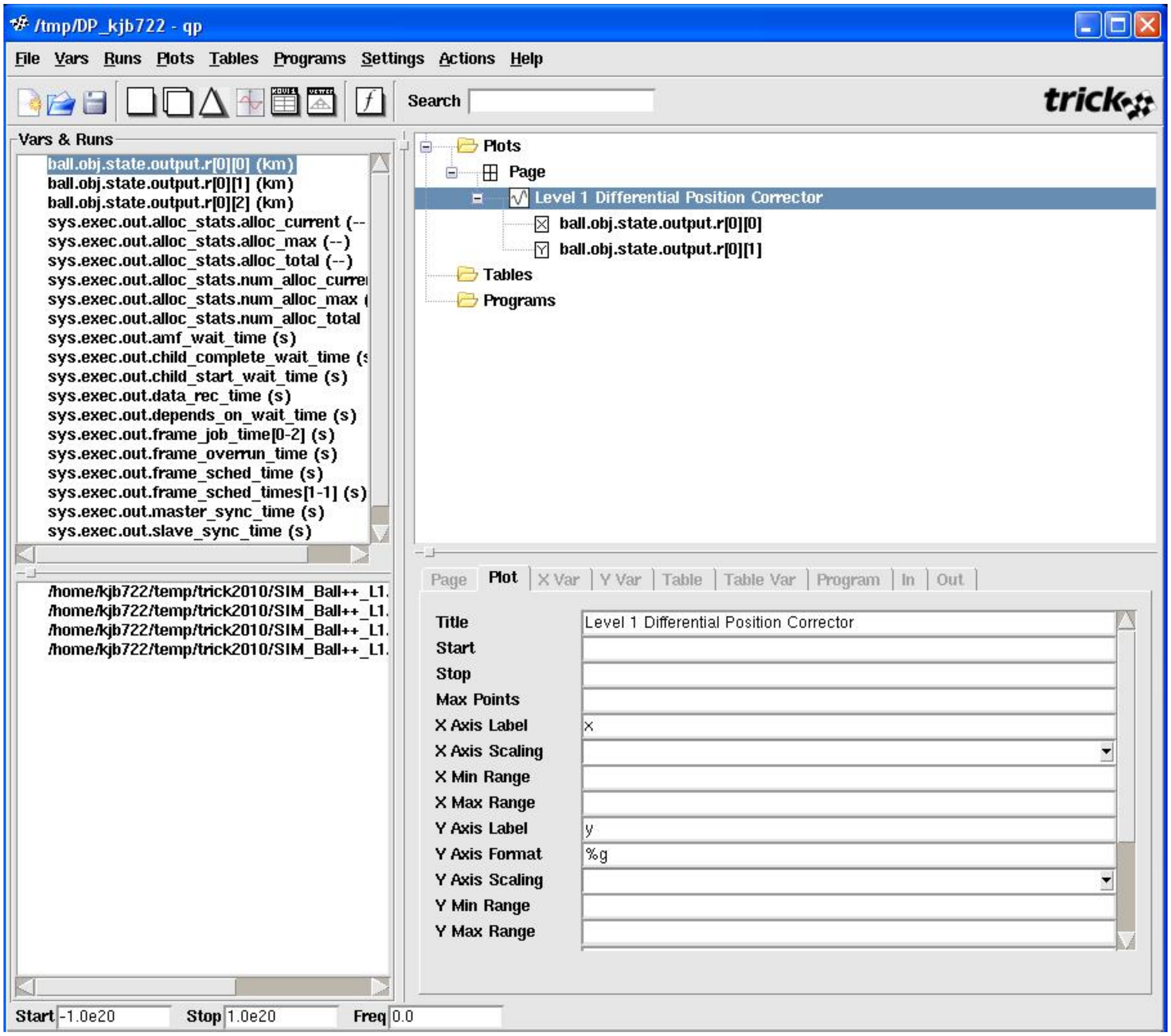
To plot the Trick Monte Carlo output curves, the four Monte Carlo output run directories are selected in the trick_dp utility, shown in Figure 10.



Start -1.0e20 Stop 1.0e20 Freq 0.0 Tolerance 1.0e-9

Figure 10: Screenshot selecting the four Monte Carlo run output directories in trick_dp

Note that the desired plot trajectories are stored in the first row of the output position matrix, as this is the reference trajectory for each run. This allows for the generation of a plot of the y position vs. the x position in Figure 11.



ball.obj.state.output.r[0][0] (km)
ball.obj.state.output.r[0][1] (km)
ball.obj.state.output.r[0][2] (km)
sys.exec.out.alloc_stats.alloc_current (--
sys.exec.out.alloc_stats.alloc_max (--)
sys.exec.out.alloc_stats.alloc_total (--)
sys.exec.out.alloc_stats.num_alloc_curre
sys.exec.out.alloc_stats.num_alloc_max (
sys.exec.out.alloc_stats.num_alloc_total
sys.exec.out.amf_wait_time (s)
sys.exec.out.child_complete_wait_time (s
sys.exec.out.child_start_wait_time (s)
sys.exec.out.data_rec_time (s)
sys.exec.out.depends_on_wait_time (s)
sys.exec.out.frame_job_time[0-2] (s)
sys.exec.out.frame_overrun_time (s)
sys.exec.out.frame_sched_time (s)
sys.exec.out.frame_sched_times[1-1] (s)
sys.exec.out.master_sync_time (s)
sys.exec.out.slave_sync_time (s)

/home/kjb722/temp/trick2010/SIM_Ball++_L1.
/home/kjb722/temp/trick2010/SIM_Ball++_L1.
/home/kjb722/temp/trick2010/SIM_Ball++_L1.
/home/kjb722/temp/trick2010/SIM_Ball++_L1.

- Plots
  - Page
    - Level 1 Differential Position Corrector
      - ball.obj.state.output.r[0][0]
      - ball.obj.state.output.r[0][1]
- Tables
- Programs

Title	Level 1 Differential Position Corrector
Start	
Stop	
Max Points	
X Axis Label	x
X Axis Scaling	
X Min Range	
X Max Range	
Y Axis Label	y
Y Axis Format	%g
Y Axis Scaling	
Y Min Range	
Y Max Range	

Start -1.0e20 Stop 1.0e20 Freq 0.0

Figure 11: Screenshot plotting the x and y components of the reference position data log for each Monte run

The plot for the qp job in Figure 11 is shown in Figure 12. Figure 13 is a plot from MATLAB which verifies the Trick output trajectories.

Page

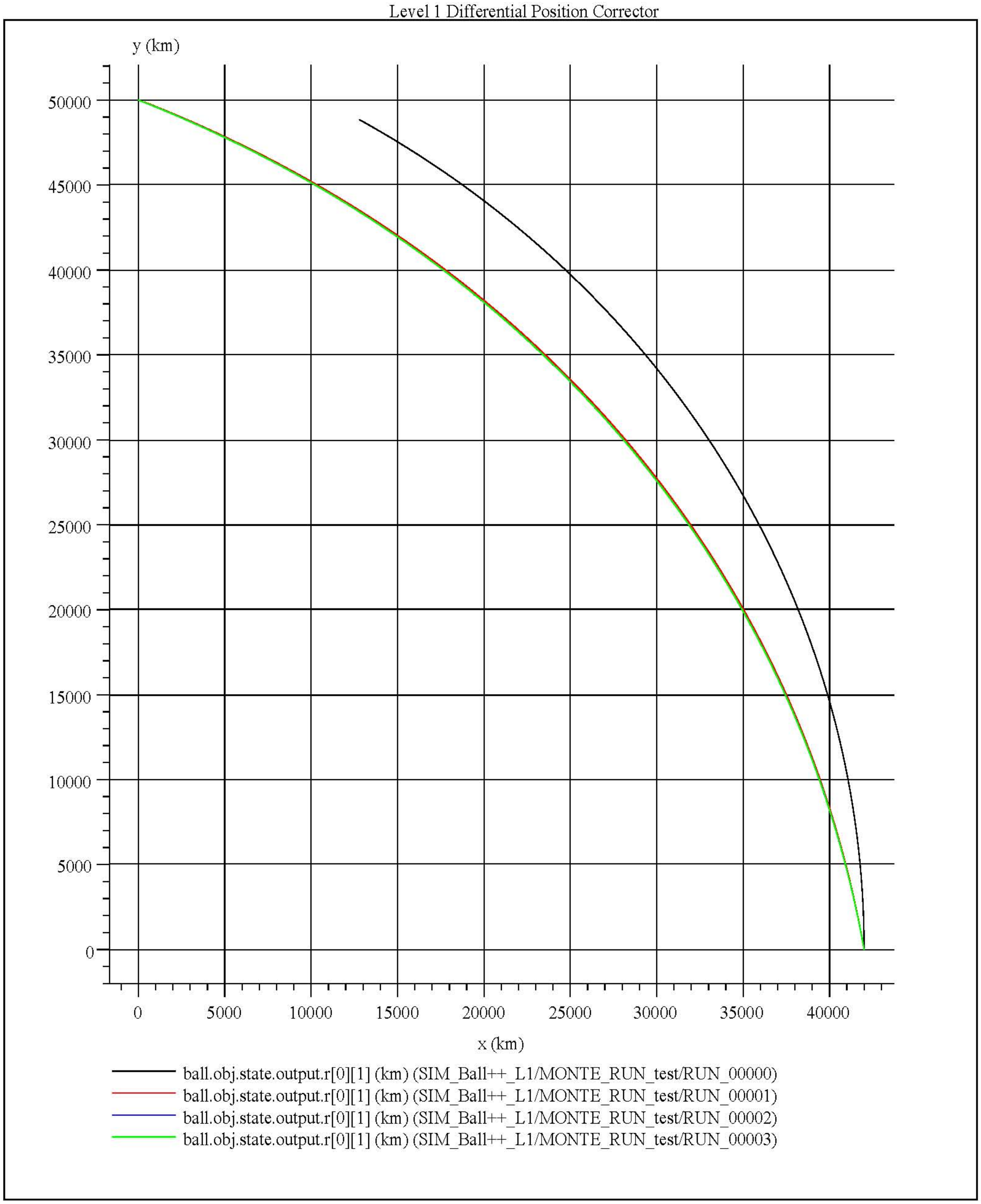


Figure 12: Trick targeting curves

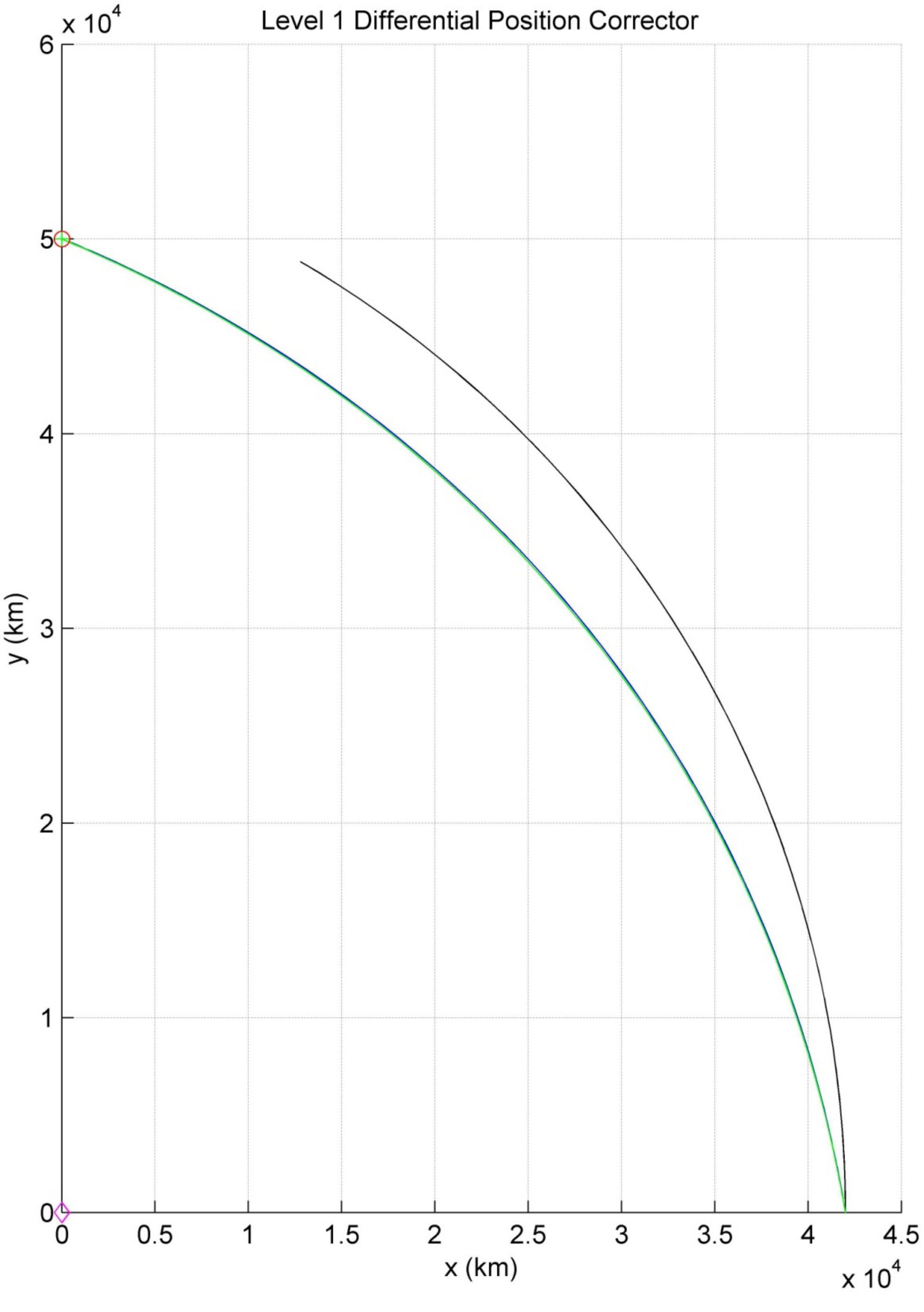
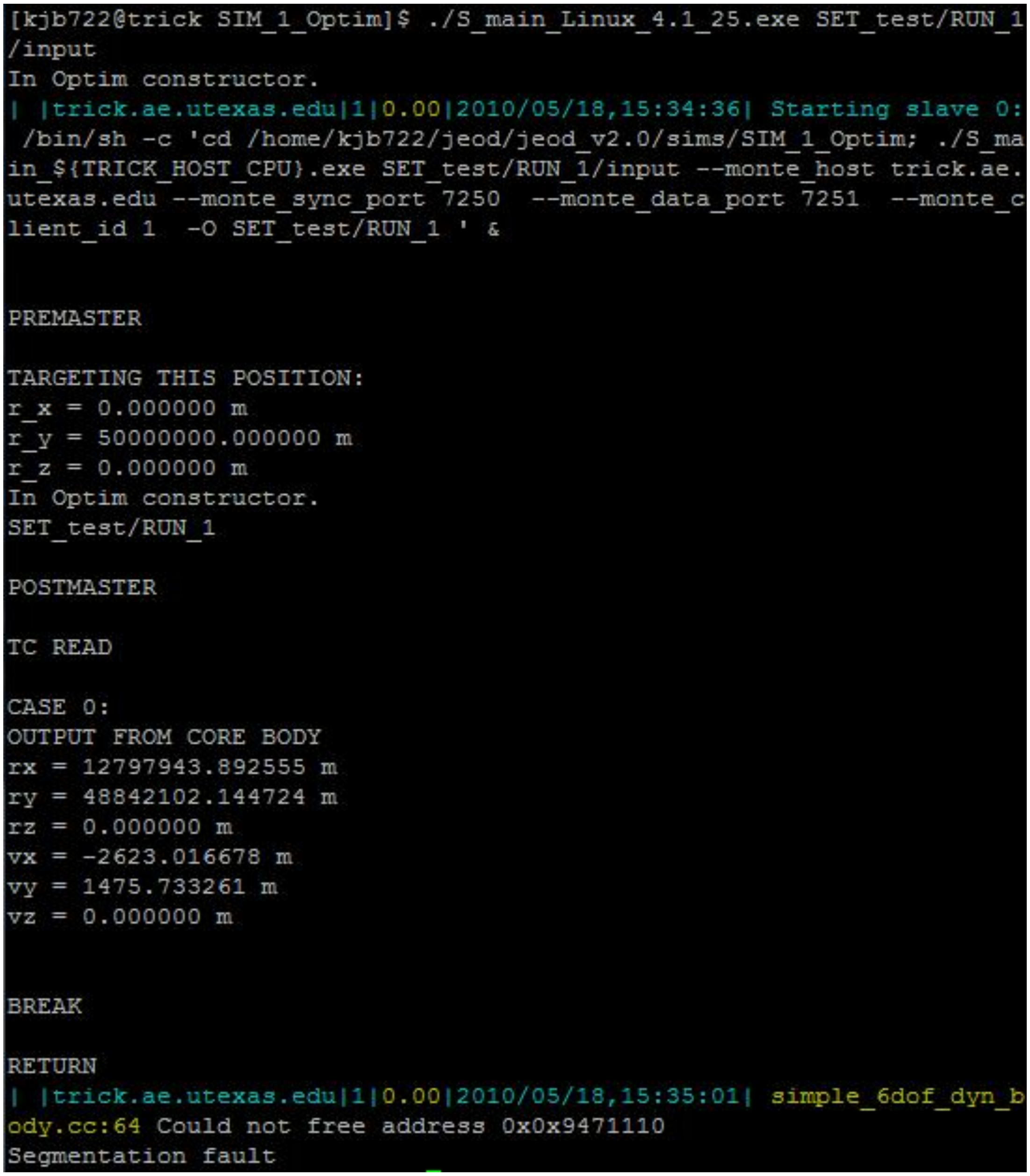


Figure 13: MATLAB targeting curves

One can see that both targeters successfully converge to the desired final location in four iterations.

5.2 JEOD

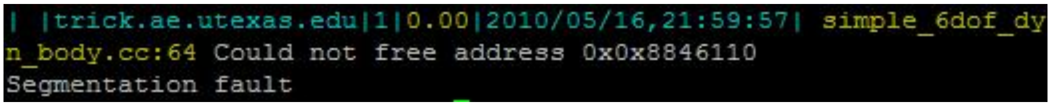
The team successfully created an optimization object in the modified `S_define` file and produced the correct trajectory for the first reference state. Comparison of the output for the first iteration in Figure 8 with the output in Figure 14 shows that both final states are the same.



```
[kjb722@trick SIM_1_Optim]$ ./S_main_Linux_4.1_25.exe SET_test/RUN_1
/input
In Optim constructor.
| |trick.ae.utexas.edu|1|0.00|2010/05/18,15:34:36| Starting slave 0:
 /bin/sh -c 'cd /home/kjb722/jeod/jeod_v2.0/sims/SIM_1_Optim; ./S_ma
in_${TRICK_HOST_CPU}.exe SET_test/RUN_1/input --monte_host trick.ae.
utexas.edu --monte_sync_port 7250  --monte_data_port 7251  --monte_c
lient_id 1  -O SET_test/RUN_1 ' &


PREMASTER

TARGETING THIS POSITION:
r_x = 0.000000 m
r_y = 50000000.000000 m
r_z = 0.000000 m
In Optim constructor.
SET_test/RUN_1

POSTMASTER

TC READ

CASE 0:
OUTPUT FROM CORE BODY
rx = 12797943.892555 m
ry = 48842102.144724 m
rz = 0.000000 m
vx = -2623.016678 m
vy = 1475.733261 m
vz = 0.000000 m


BREAK

RETURN
| |trick.ae.utexas.edu|1|0.00|2010/05/18,15:35:01| simple_6dof_dyn_b
ody.cc:64 Could not free address 0x0x9471110
Segmentation fault
```

Figure 14: JEOD Targeter Output

However, a memory error occurred when the Monte Carlo framework attempted to perform the master loop iteration.



```
| |trick.ae.utexas.edu|1|0.00|2010/05/16,21:59:57| simple_6dof_dy
n_body.cc:64 Could not free address 0x0x8846110
Segmentation fault
```

Figure 15: Memory Error for JEOD integration

Apparently, `simple_6dof_dyn_body.cc` attempted to delete one of its objects but the optimization framework did not allow it. Commenting out all deletions in the `Simple6DofDynBody` destructor successfully eliminated the address error but did not get rid of the segmentation fault.

There are some important points to make about the output in Figure 14. First, the printouts from the pre and post masters let the user know that the Monte Carlo job is indeed functioning correctly. Also, the master has read the slave's results successfully, as indicated by the output in the post master of the final state of the vehicle. Then, since the `RETURN` statement is printed to the screen, the Monte Carlo framework leaves the post master without error. After the framework leaves the post master however, it does not successfully loop back to the pre-master before the failure of the simulation. Because of this, the address error and the segmentation fault appear to occur in the background interaction between the Monte Carlo framework and JEOD.

Some solutions were investigated for working around the segmentation fault, but none proved successful. Also, commenting out the memory error caused by the destructor is contrary to the goal of leaving JEOD unmodified when integrating with the targeter. Due to the relative inexperience with JEOD, the team struggled in determining an appropriate fix to the address freeing issue without modifying the destructor.

6.0 CONCLUSION

6.1 Discussion

A successful method for targeting a desired final position from an initial state has been presented using the Trick optimization framework. This met the first goal of the project. The caveat to this is that the satellite dynamical model was user specified and extremely simplified. An attempt was made to implement more complex models provided by JEOD, but wrapping the targeter around one of the Tutorial simulations proved unsuccessful. This means that the second goal was not met. However, the attempt is documented for future study and discussion of the proposed algorithm for interfacing a targeter with a "black box" physical model included. Also, because of the presented errors, the switch statement method outlined in section 4.2 could not be fully verified in run time. This study is not prepared to draw any final conclusions about the compatibility of JEOD and the Monte Carlo framework.

6.2 Recommendations for Future Work

The targeting algorithm presents several avenues for future development, some of which have been explored over the course of the semester.

First, the team has spent significant time working with the JEOD simulation tool. Steps have been taken towards understanding the obstacles involved with integrating this tool into the simulation portion of the targeting algorithm. A JEOD implementation would allow for the exploration of trajectory perturbations such as multi-body dynamics, atmospheric drag, and solar radiation pressure. Such an increase in fidelity would better the accuracy of results and improve usefulness to the community.

Considering the failure of this initial, simple targeting scheme, a feasibility analysis devoted to the JEOD/Monte Carlo interaction is likely the next course of action. For a future feasibility study, someone more familiar with JEOD will likely need to serve in a support role. This project did not request such assistance because the issues were discovered at the very end of the semester and time had run out to investigate further.

Second, the team has looked into developing a targeter which does not require a constant simulation time. The model has been implemented in MATLAB but porting this code to Trick requires further research. The ability to make the time a target variable might allow for optimization of the initial impulse burn beyond the solutions suggested by the current implementation.

Additionally, one of the strengths of Trick is its ability to act as a backbone of graphical or real time human in the loop simulations. The targeter in its present form would not benefit from the latter, but developing a visualization component, similar to those employed by STK and Copernicus, would improve usability and the effectiveness of conveying results.

Lastly, in the process of further development, naming conventions of files and classes shall be improved to be more representative of the underlying code. This will help to eschew confusion with previous code sets and capabilities.

7.0 REFERENCES

[1] Marchand, B., Howell, K., and Wilson, R., " Improved Corrections Process for Constrained Trajectory Design in the n-Body Problem," *Journal of Spacecraft and Rockets*, Vol. 44, No. 4, 2007, pp. 884-897.

[2] Moles, S., Tschirhart, Z., and Tseung, B., “Optimization in Trick,” 2009, pp. 1-40.

[3] Bate, R., Mueller, D., and White, J., "Fundamentals of Astrodynamics," Dover Publications, New York, 1st ed., 1971, pp. 122–131.

[4] Corless, M. and Frazho, A., *Linear Systems and Control: An Operator Perspective*, CRC, 2003, pp.313–314.

[5] Vetter, K. and Hua, G., "The Trick Users Guide Trick 2005.7.0 Release," 2006, pp. 1-169.

[6] Jackson, A. and Thebeau, C., "JSC Engineering Orbital Dynamics (JEOD) Top Level Document," JSC-61777-docs, 2009, pp. 1-49.

8.0 APPENDICES

There is one important item to note in looking at the code included in the Appendices. While the code directory is called `Optim++` and the simulation object is called `optimizer`, nothing is actually being optimized in the presented example. Instead, the method simply targets a desired final spatial location given some arbitrary initial state. The optimization names were preserved from previous projects because the Trick Monte Carlo Optimization framework is used here to accomplish the targeting procedure.

Appendix A — *Finite Differencing Code*

Ball++/L1/include/Ball.dd

```
/**************************** TRICK HEADER *************************
PURPOSE:
    (Ball model parameter default init. data.)
REFERENCE:
    (((Bailey, R.W, and Paddock, E.J.)
      (Trick Simulation Environment) (NASA:JSC #37943)
      (JSC/Engineering Directorate/Automation, Robotics and Simulation
Division)
      (March 1997)))
ASSUMPTIONS AND LIMITATIONS:
    ((2 dimensional space)
     (Constant Force))
PROGRAMMERS:
    (((Edwin Z. Crues) (Titan Systems Corp.) (Feb 2002) (C++ Ball
Model)))
    ((Kyle Brill, Chun-Yi Wu, Victor Rodriguez, Harsh Shah)(UT
Austin)(May 2010)))
*******************************************************************/

Ball.state.input.mass {kg} = 10.0 ;

Ball.state.input.r[0] {km} = 42000.0 ;
Ball.state.input.r[1] {km} = 0.0 ;
Ball.state.input.r[2] {km} = 0.0 ;

Ball.state.input.v[0] {km/s} = 0.0 ;
Ball.state.input.v[1] {km/s} = 3.5 ;
Ball.state.input.v[2] {km/s} = 0.0 ;

Ball.state.input.pertr[0] {km} = .01;
Ball.state.input.pertr[1] {km} = .01;
Ball.state.input.pertr[2] {km} = .01;
Ball.state.input.pertv[0] {km/s} = .0001;
Ball.state.input.pertv[1] {km/s} = .0001;
Ball.state.input.pertv[2] {km/s} = .0001;

Ball.state.input.mu {km3/s2} = 398600.0;
```

Ball++/L1/include/Ball.hh

```
/***************************** TRICK HEADER *************************
PURPOSE:
    (Ball model EOM state parameter definition.)
REFERENCES:
    (((Bailey, R.W, and Paddock, E.J.)
      (Trick Simulation Environment) (NASA:JSC #37943)
      (JSC/Engineering Directorate/Automation, Robotics and Simulation
Division)
      (March 1997)))
ASSUMPTIONS AND LIMITATIONS:
     (Translational EOM only))
LIBRARY DEPENDENCY:
    ((Ball.o)
     (BallStateDeriv.o)
     (BallStateInit.o)
     (BallStateInteg.o)
     (BallForceField.o))
PROGRAMMERS:
    (((Robert W. Bailey) (Sweet Systems Inc) (March 1997) (Tutorial
Lesson 1))
     ((Edwin Z. Crues)(Titan Systems Corp.)(Jan 2002)(Crude C++
translation)))
     ((Kyle Brill, Chun-Yi Wu, Victor Rodriguez, Harsh Shah)(UT
Austin)(May 2010)))
*******************************************************************/


#ifndef _BALL_HH_
#define _BALL_HH_

// Trick include files.
#include "sim_services/include/integrator.h"

/* Model include files. */
#include "BallState.hh"

class Ball {

  public:

   // Default constructor and destructor.
   Ball();
   ~Ball();

   // Initialization functions.
   int state_init();

   // Derivative class jobs.
   int state_deriv();
```

```
   // State Transition Matrix calculator.
   int calc_phi(double phi[6][6]);

   // Integration class jobs.
   int state_integ( INTEGRATOR * integ );

   // Trick requires all logged data to be public.
   BallState state; /* -- Ball state object. */

};

#endif /* _BALL_HH_ */
```

Ball++/L1/include/ball_integ.d

```
/***************************** TRICK HEADER *************************
PURPOSE:
    (Ball model state integrator default initialization data.)
REFERENCE:
    (((Bailey, R.W, and Paddock, E.J.)
      (Trick Simulation Environment) (NASA:JSC #37943)
      (JSC/Engineering Directorate/Automation, Robotics and Simulation
Division)
      (March 1997)))
ASSUMPTIONS AND LIMITATIONS:
    ((3 dimensional space))
PROGRAMMERS:
    ((Kyle Brill, Chun-Yi Wu, Victor Rodriguez)(UT Austin)(May 2010)))
*******************************************************************/

#define NUM_STEP        12  /* use up to 12 intermediate steps:
                               8th order RK Fehlberg */
#define NUM_VARIABLES   42   /* x,y,z position state and x,y,z
velocity state */

INTEGRATOR.state = alloc(NUM_VARIABLES) ;
INTEGRATOR.deriv = alloc(NUM_STEP) ;
INTEGRATOR.state_ws = alloc(NUM_STEP) ;
for (int kk = 0 ; kk < NUM_STEP ; kk++ ) {
    INTEGRATOR.deriv[kk] = alloc(NUM_VARIABLES) ;
    INTEGRATOR.state_ws[kk] = alloc(NUM_VARIABLES) ;
}
INTEGRATOR.num_state    = NUM_VARIABLES ;
INTEGRATOR.option       = Runge_Kutta_4 ;    /* 4th order Runge Kutta
*/
INTEGRATOR.init         = True ;
INTEGRATOR.first_step_deriv = Yes ;

#undef NUM_STEP
#undef NUM_VARIABLES
```

Ball++/L1/include/BallState.hh

```
/***************************** TRICK HEADER **************************
PURPOSE:
    (Ball model state parameter definition.)
REFERENCE:
    (((Bailey, R.W, and Paddock, E.J.)
      (Trick Simulation Environment) (NASA:JSC #37943)
      (JSC/Engineering Directorate/Automation, Robotics and Simulation
Division)
      (March 1997)))
ASSUMPTIONS AND LIMITATIONS:
     (Always toward a stationary point))
PROGRAMMERS:
    (((Robert W. Bailey) (Sweet Systems Inc) (March 1997) (Tutorial
Lesson 1))
     ((Edwin Z. Crues)(Titan Systems Corp.)(Jan 2002)(Crude C++
translation)))
     ((Kyle Brill, Chun-Yi Wu, Victor Rodriguez)(UT Austin)(May
2010)))
*********************************************************************/

#ifndef _BALL_STATE_HH_
#define _BALL_STATE_HH_

class BallStateInput {
 public:
  double mass;             /* *i kg Total mass. */
  double mu;            /* *i (km3/s2) gravitational parameter*/
  double r[3];        /* *i (km) position vector */
  double v[3];             /* *i (km/s) velocity vector */
  double pertr[3];         /* *i (km) position perturbation vector */
  double pertv[3];         /* *i (km/s) velocity perturbation vector
*/
};

class BallStateOutput {
 public:
  double r[7][3];                /* *o (km) position states matrix */
  double v[7][3];                /* *o (km/s) velocity states matrix */
  double a[7][3];          /* *o (km/s2) XYZ accelerations matrix */
};

class BallState {

  public:
   /* Member data. */
   BallStateInput  input;  /* -- User inputs    */
   BallStateOutput output; /* -- User outputs.  */
};

#endif /* _BALL_STATE_HH_ */
```

Ball++/L1/src/Ball.cpp

```
/***************************** TRICK HEADER *************************
PURPOSE:
    (Ball::Ball ball object constructor.)
    (Ball::calc_phi state transition matrix calculator.)
REFERENCE:
    (((Bailey, R.W, and Paddock, E.J.)
      (Trick Simulation Environment) (NASA:JSC #37943)
      (JSC/Engineering Directorate/Automation, Robotics and Simulation
Division)
      (March 1997)))
ASSUMPTIONS AND LIMITATIONS:
    ((3 dimensional space)
     (X-axis is horizontal and positive to the right)
     (Y-axis is vertical and positive up)
CLASS:
    (N/A)
LIBRARY DEPENDENCY:
    ((Ball.o))
PROGRAMMERS:
    (((Robert W. Bailey) (Sweet Systems Inc) (March 1997) (Tutorial
Lesson 1))
     ((Edwin Z. Crues)(Titan Systems Corp.)(Jan 2002)(Crude C++
translation)))
     ((Kyle Brill, Chun-Yi Wu, Victor Rodriguez, Harsh Shah)(UT
Austin)(May 2010)))
*******************************************************************/

/* System include files. */
#include <iostream>

/* Model include files. */
#include "../include/Ball.hh"
#include "../include/BallState.hh"

// Default consructor.
Ball::Ball() /* RETURN: -- None. */
{
   // Print out constructor message.
   printf("In Ball constructor.\n");
}
// Destructor.
Ball::~Ball() /* RETURN: -- None. */
{
   // Print out destructor message.
   printf("In Ball destructor.\n");
}
```

```
/* ENTRY POINT */
int Ball::calc_phi(double phi[6][6]) /* RETURN: -- Always return zero.
*/
{

   /* PERTURBATION VECTOR */
   double perts[6];
   /* OUTPUT STATES VECTOR */
   double outputs [42];

   perts[0] = this->state.input.pertr[0];
   perts[1] = this->state.input.pertr[1];
   perts[2] = this->state.input.pertr[2];
   perts[3] = this->state.input.pertv[0];
   perts[4] = this->state.input.pertv[1];
   perts[5] = this->state.input.pertv[2];

   /* CONVERT OUTPUT STATE MATRICES TO OUTPUT STATES VECTOR */
   for ( int n = 0 ; n < 7 ; n++ )
   {
       outputs[6*n+0] = this->state.output.r[n][0];
       outputs[6*n+1] = this->state.output.r[n][1];
       outputs[6*n+2] = this->state.output.r[n][2];
       outputs[6*n+3] = this->state.output.v[n][0];
       outputs[6*n+4] = this->state.output.v[n][1];
       outputs[6*n+5] = this->state.output.v[n][2];
   }

   /* CALCULATE STATE TRANSITION MATRIX -- phi */
   for ( int i = 0 ; i < 6 ; i++ )
   {
       for ( int j = 0 ; j < 6 ; j++ )
       {
               phi[j][i] = ( outputs[j+6*(i+1)] - outputs[j] ) /
perts[i] ;
       }
   }

   /* RETURN */
   return(0);

}
```

Ball++/L1/src/BallStateDeriv.cpp

```
/*************************** TRICK HEADER *************************
PURPOSE:
    (Ball::state_deriv solves for the Ball accelerations)
REFERENCE:
    (((Bailey, R.W, and Paddock, E.J.)
      (Trick Simulation Environment) (NASA:JSC #37943)
      (JSC/Engineering Directorate/Automation, Robotics and Simulation
Division)
      (March 1997)))
ASSUMPTIONS AND LIMITATIONS:
    ((3 dimensional space)
     (X-axis is horizontal and positive to the right)
     (Y-axis is vertical and positive up)
     (derivative of position already exists as velocity vector)
CLASS:
    (derivative)
LIBRARY DEPENDENCY:
    ((BallStateDeriv.o))
PROGRAMMERS:
    (((Robert W. Bailey) (Sweet Systems Inc) (March 1997) (Tutorial
Lesson 1))
     ((Edwin Z. Crues)(Titan Systems Corp.)(Jan 2002)(Crude C++
translation)))
     ((Victor Rodriguez, Harsh Shah, Kyle Brill)(UT Austin)(May
2010)))
*******************************************************************/
/* Trick include files. */
#include "trick_utils/math/include/vector_macros.h"
/* Model include files. */
#include "../include/Ball.hh"
/* ENTRY POINT */
int Ball::state_deriv() /* RETURN: -- Always return zero. */
{
   /* GET SHORTHAND NOTATION FOR DATA STRUCTURES */
   BallStateInput  * state_in   = &(this->state.input);
   BallStateOutput * state_out  = &(this->state.output);
   /* SOLVE FOR THE X AND Y ACCELERATIONS OF THE BALL */
   for(int i = 0; i<7; i++)
   {
      double mag = V_MAG(state_out->r[i]);
      state_out->a[i][0] =  -state_in->mu * state_out->r[i][0] /
pow(mag,3) ;
      state_out->a[i][1] =  -state_in->mu * state_out->r[i][1] /
pow(mag,3) ;
      state_out->a[i][2] =  -state_in->mu * state_out->r[i][2] /
pow(mag,3);
   }
   /* RETURN */
   return(0);
}
```

Ball++/L1/src/BallStateInit.cpp

```
/**************************** TRICK HEADER **************************
PURPOSE:
    (Ball::state_init passes the input state vectors to the output
state matrices.)
REFERENCE:
    (((Bailey, R.W, and Paddock, E.J.)
      (Trick Simulation Environment) (NASA:JSC #37943)
      (JSC/Engineering Directorate/Automation, Robotics and Simulation
Division)
      (March 1997)))
ASSUMPTIONS AND LIMITATIONS:
    ((3 dimensional space)
     (Positive X is horizontal to the right)
     (Positive Y is vertical and up))
CLASS:
    (initialization)
LIBRARY DEPENDENCY:
    ((BallStateInit.o))
PROGRAMMERS:
    (((Robert W. Bailey) (Sweet Systems Inc) (March 1997) (Tutorial
Lesson 1))
     ((Edwin Z. Crues)(Titan Systems Corp.)(Jan 2002)(Crude C++
translation)))
     ((Chun-Yi Wu, Kyle Brill, Victor Rodriguez)(UT Austin)(May 2010))
*********************************************************************/

/* Model include files. */
#include "../include/Ball.hh"

/* ENTRY POINT */
int Ball::state_init() /* RETURN: -- Always return zero. */
{

   /* GET SHORHAND NOTATION FOR DATA STRUCTURES */
   BallStateInput  * state_in  = &(this->state.input);
   BallStateOutput * state_out = &(this->state.output);

   /* TRANSFER INPUT POSITION AND VELOCITY VECTORS TO
      OUPUT STATES MATRICES */
   for ( int n = 0 ; n < 7 ; n++ )
   {
      state_out->r[n][0] = state_in->r[0];  /* X state */
      state_out->r[n][1] = state_in->r[1];  /* Y state */
      state_out->r[n][2] = state_in->r[2];  /* Z state */

      state_out->v[n][0] = state_in->v[0];
      state_out->v[n][1] = state_in->v[1];
      state_out->v[n][2] = state_in->v[2];
   }
```

```
   /* ADD THE APPROPRIATE PERTURBATIONS TO THE NECESSARY
      STATES IN THE OUTPUT STATES MATRICES */
   for ( int i = 0 ; i < 3 ; i++ )
   {
      state_out->r[i+1][i] += state_in->pertr[i];
      state_out->v[i+4][i] += state_in->pertv[i];
   }

   /* RETURN */
   return( 0 );

}
```

Ball++/L1/src/BallStateInteg.cpp

```
/**************************** TRICK HEADER *************************
PURPOSE:
    (Ball::state_integ performs the following:
        -loads the position states into the INTEGRATOR state ws arrays
        -loads the velocity states into the INTEGRATOR state
derivative ws
        -loads the velocity states into the INTEGRATOR state ws arrays
        -loads the acceleration states into the INTEGRATOR state
derivative ws
        -calls the TRICK state integration service
        -unloads the new states from the INTEGRATOR ws arrays
REFERENCE:
    (((Bailey, R.W, and Paddock, E.J.)
      (Trick Simulation Environment) (NASA:JSC #37943)
      (JSC/Engineering Directorate/Automation, Robotics and Simulation
Division)
      (March 1997)))
ASSUMPTIONS AND LIMITATIONS:
    ((3 dimensional space)
     (integrate accel to pos as two first order diffEQs))
CLASS:
    (integration)
LIBRARY DEPENDENCY:
    ((BallStateInteg.o))
PROGRAMMERS:
    (((Robert W. Bailey) (Sweet Systems Inc) (March 1997) (Tutorial
Lesson 1))
     ((Edwin Z. Crues)(Titan Systems Corp.)(Jan 2002)(Crude C++
translation)))
     ((Kyle Brill, Chun-Yi Wu, Victor Rodriguez)(UT Austin)(May
2010)))
*******************************************************************/

/* System include files. */
#include <stdio.h>

/* Trick include files. */
#include "sim_services/include/integrator.h"

/* Model include files. */
#include "../include/Ball.hh"

/* ENTRY POINT */
int Ball::state_integ(  /* RETURN: -- Integration multi-step id.
*/
   INTEGRATOR * integ ) /* INOUT:  -- Trick state integration
parameters. */
{
```

```
   /* GET SHORTHAND NOTATION FOR DATA STRUCTURES */
   BallStateOutput * state_out = &(this->state.output);

   /* LOAD THE POSITION AND VELOCITY STATES */
   for ( int n = 0 ; n < 7 ; n++ )
   {
        integ->state[n*6+0] = state_out->r[n][0];
        integ->state[n*6+1] = state_out->r[n][1];
        integ->state[n*6+2] = state_out->r[n][2];
        integ->state[n*6+3] = state_out->v[n][0];
        integ->state[n*6+4] = state_out->v[n][1];
        integ->state[n*6+5] = state_out->v[n][2];
   }

   /* LOAD THE POSITION AND VELOCITY STATE DERIVATIVES */
   for ( int n = 0 ; n < 7 ; n++ )
   {
      integ->deriv[integ->intermediate_step][n*6+0] = state_out->v[n][0];
     integ->deriv[integ->intermediate_step][n*6+1] = state_out->v[n][1];
     integ->deriv[integ->intermediate_step][n*6+2] = state_out->v[n][2];
     integ->deriv[integ->intermediate_step][n*6+3] = state_out->a[n][0];
     integ->deriv[integ->intermediate_step][n*6+4] = state_out->a[n][1];
      integ->deriv[integ->intermediate_step][n*6+5] = state_out->a[n][2];
   }

   /* CALL THE TRICK INTEGRATION SERVICE */
   integrate( integ );

   /* UNLOAD THE NEW POSITION AND VELOCITY STATES */
   for ( int n = 0 ; n < 7 ; n++ )
   {
      state_out->r[n][0] = integ->state_ws[integ->intermediate_step][n*6+0];
     state_out->r[n][1] = integ->state_ws[integ->intermediate_step][n*6+1];
     state_out->r[n][2] = integ->state_ws[integ->intermediate_step][n*6+2];
     state_out->v[n][0] = integ->state_ws[integ->intermediate_step][n*6+3];
     state_out->v[n][1] = integ->state_ws[integ->intermediate_step][n*6+4];
     state_out->v[n][2] = integ->state_ws[integ->intermediate_step][n*6+5];
   }


   /* RETURN */
   return( integ->intermediate_step );

}
```

Appendix B — *Targeting Code*

Ball++/Optim++/include/Optimization.dd

```
/************************* TRICK HEADER *****************************
PURPOSE:        (Optimization Default Data)
*******************************************************************/

Optimization.optim_v[0] {km/s} = 0.0 ;
Optimization.optim_v[1] {km/s} = 0.0 ;
Optimization.optim_v[2] {km/s} = 0.0 ;

Optimization.deltav[0] {km/s} = 0.0 ;
Optimization.deltav[1] {km/s} = 0.0 ;
Optimization.deltav[2] {km/s} = 0.0 ;

Optimization.r_des[0]  {km}   = 0.0;
Optimization.r_des[1]  {km}   = 50000.0;
Optimization.r_des[2]  {km}   = 0.0;
```

Ball++/Optim++/include/Optimization.hh

```
/************************* TRICK HEADER ******************************
PURPOSE:
      ( Header for Optimization class )
LIBRARY DEPENDENCY:
      ((Optimization.o)
       (OptimInit.o)
       (OptimPreJobs.o)
       (OptimPostJobs.o))
*******************************************************************/

#ifndef _OPTIMIZATION_HH_
#define _OPTIMIZATION_HH_

#include "../../L1/include/Ball.hh"

class Optimization
{
  public:
   // Constructor and Destructor
   Optimization();
   ~Optimization();

   // Optimization variables
   double optim_v[3]; /* (km/s) optimal initial velocity */
   double deltav[3];  /* (km/s) change in initial velocity*/
   double rf[3]; /* (km) actual final position*/
```

```
    double phi[6][6];  /* state transition matrix*/
    double r_des[3]; /* (km) desired final position*/
    double r0[3]; /* (km) the initial position of the first run */
    double v0[3]; /* (km/s) the initial velocity of the first run */
    int counter; /* iteration counter */

    // Optimization Initialization
    int optim_init();

    // The "pre" job function. This contains the optimization
    // algorithm and stopping condtions.
    int ball_pre_master(Ball* B);

    // The "post" job functions. These contain the TCP/IP
    // connection that the master/slave framework uses.
    int ball_post_master();
    int ball_post_slave(Ball* B);
};

#endif
```

Ball++/Optim++/src/Optimization.cpp

```
/************************* TRICK HEADER ***************************
PURPOSE:
    (Optimization:Optimization optimizer object constructor.)
CLASS:
    (N/A)
LIBRARY DEPENDENCY:
    ((Optimization.o))
PROGRAMMERS:
    ((Kyle Brill, Chun-Yi Wu, Victor Rodriguez)(UT Austin)(May 2010)))
*******************************************************************/

/* System include files. */
#include <iostream>

/* Model include files. */
#include "../include/Optimization.hh"

// Default constructor
Optimization::Optimization() /* RETURN: -- None. */
{
   printf("In Optim constructor.\n");
}

// Destructor
Optimization::~Optimization() /* RETURN: -- None. */
{
   printf("In Optim destructor.\n");
}
```

Ball++/Optim++/src/OptimInit.cpp

```
/**************************** TRICK HEADER *************************
PURPOSE:
    (Optimization::state_init initializes all necessary variables
CLASS:
    (monte_master_init)
LIBRARY DEPENDENCY:
    ((OptimInit.o))
PROGRAMMERS:
    (((Chun-Yi Wu, Kyle Brill, Victor Rodriguez)(UT Austin)(May 2010))
*******************************************************************/

#include "../include/Optimization.hh"

/* ENTRY POINT */
int Optimization::optim_init() /* RETURN: -- Always return zero. */
{
   /* INITIALIZE ITERATION COUNTER */
   counter = 0;

   /* RETURN */
   return (0) ;
}
```

Ball++/Optim++/src/OptimPreJobs.cpp

```
/********************** TRICK HEADER *******************************
PURPOSE:
    (Optimization::ball_pre_master contains the targeting algorithm
     and end condition)
CLASS:
    (monte_master_pre)
LIBRARY DEPENDENCY:
    ((OptimPreJobs.o))
PROGRAMMERS:
    (((Chun-Yi Wu, Kyle Brill, Victor Rodriguez, Field Manar)(UT
Austin)(May 2010))
*******************************************************************/

/* System include files. */
#include <iostream>

/* Trick include files. */
#include "sim_services/include/exec_proto.h"
#include "trick_utils/math/include/trick_math.h"
#include "trick_utils/math/include/vector_macros.h"

/* Model include files. */
#include "../include/Optimization.hh"
#include "../../L1/include/Ball.hh"

/* ENTRY POINT */
int Optimization::ball_pre_master(Ball *B) /* RETURN: -- Always return
zero. */
{
   /* INCREMENT ITERATION COUNTER */
   this->counter++ ;

   double B_mat[3][3];
   double B_inv[3][3];
   double error[3];

   if ( this->counter > 1 )
   {
      /* CALCULATE THE DIFFERENCE BETWEEN THE DESIRED FINAL POSITION
         AND THE ACTUAL FINAL POSITION */
      V_SUB( error , this->r_des , this->rf );

      /* ABSOLUTE VALUE OF THE MAXIMUM ERROR OF ALL POSITION
COMPONENTS */
      double max = error[0];
      if ( -error[1] > max )
         max = -error[1];
      if ( error[1] > max )
         max = error[1];
      if ( -error[0] > max )
```

```
         max = -error[0];
      if ( error[2] > max )
         max = error[2];
      if ( -error[2] > max )
         max = -error[2];

      printf("\nIteration #: %d\n", this->counter-1);
      printf("\nFinal Position:\nr_x = %f km\nr_y = %f km\nr_z = %f
km\n",rf[0],rf[1],rf[2]);

      /* STOPPING CONDITION IS MAX ERROR < .00001 km OR 20 ITERATIONS
*/
      if( max < 0.00001 || this->counter == 20)
      {
         printf("\n********************\n");
         printf("The change in velocity necessary to reach the desired
final location\n\n");
         printf("r_des_x = %f km\nr_des_y = %f km\nr_des_z = %f
km\n",r_des[0],r_des[1],r_des[2]);
         printf("\nfrom the initial state\n\nr_x0 = %f km\nr_y0 = %f
km\nr_z0 = %f km\nv_x0 = %f km/s\nv_y0 = %f km/s\nv_z0 = %f
km/s\n",r0[0],r0[1],r0[2],v0[0],v0[1],v0[2]);
         printf("\nis\n\ndeltav_x = %f km/s\ndeltav_y = %f
km/s\ndeltav_z = %f km/s\n",optim_v[0]-v0[0],optim_v[1]-
v0[1],optim_v[2]-v0[2]);
         printf("\nThe targeter took %d iterations\n",this->counter-
1);
         printf("********************\n\n\n");

         /* TERMINATES THIS FUNCTION ONCE THE END CONDITION IS REACHED
*/
         exec_terminate("ball_pre_master","End Condition reached.");
      }
      else
      {
         printf("\nContinuing algorithm...end condition not
met.\n\n");
      }

      B->state.input.v[0] = this->optim_v[0] ;
      B->state.input.v[1] = this->optim_v[1] ;
      B->state.input.v[2] = this->optim_v[2] ;
   }
   else
   {
      /* STORE FIRST INTIAL STATE SO WE CAN SUGGEST A DELTA V
         AT THE END OF THE ALGORITHM */
      this->r0[0] = B->state.input.r[0];
      this->r0[1] = B->state.input.r[1];
      this->r0[2] = B->state.input.r[2];
      this->v0[0] = B->state.input.v[0];
      this->v0[1] = B->state.input.v[1];
```

```
      this->v0[2] = B->state.input.v[2];

      /* RETURN -- this is the first run, nothing else left to do*/
      return (0);
   }

   /* RETURN */
   return (0) ;

}
```

Ball++/Optim++/src/OptimPostJobs.cpp

```
/************************* TRICK HEADER *****************************
PURPOSE:
    (Optimization::ball_post_master read slave results via
     TCP/IP Comm and calculate state transition matrix)
LIBRARY DEPENDENCY:
    ((OptimPostJobs.o))
PROGRAMMERS:
    (((Chun-Yi Wu, Kyle Brill, Victor Rodriguez, Field Manar)(UT
Austin)(May 2010))
*******************************************************************/

/* System include files. */
#include <iostream>

/* Trick include files. */
#include "sim_services/include/executive.h"
#include "sim_services/include/exec_proto.h"
#include "trick_utils/math/include/trick_math.h"
#include "trick_utils/math/include/vector_macros.h"

/* Model include files. */
#include "../include/Optimization.hh"
#include "Ball++/L1/include/Ball.hh"

/* ENTRY POINT */
int Optimization::ball_post_master() /* RETURN: -- Always return zero.
*/
{
   double B_mat[3][3], B_inv[3][3], error[3], deltav[3];

   Ball B_curr ;
   EXECUTIVE* E ;
   E = exec_get_exec() ;

   /* READ SLAVE'S RESULTS */
   tc_read( &E->monte.work.data_conn, (char*) &B_curr, sizeof(Ball) );

   /* STORE FINAL OUTPUT POSITION VECTOR */
   for ( int n = 0 ; n < 3 ; n++ )
   {
      this->rf[n] = B_curr.state.output.r[0][n];
   }

   /* CALCULATE THE STATE TRANSITION MATRIX FOR THE CURRENT BALL
OBJECT */
   B_curr.calc_phi((this->phi));

   for(int i = 0; i <3; i++){
        for(int j = 0; j<3; j++){
             B_mat[i][j] = this->phi[i][j+3];
```

```
        }
    }

    V_SUB(error, this->r_des, this->rf);
    dm_invert(B_inv, B_mat);
    MxV(deltav, B_inv, error);

    this->optim_v[0] = B_curr.state.input.v[0] + deltav[0];
    this->optim_v[1] = B_curr.state.input.v[1] + deltav[1];
    this->optim_v[2] = B_curr.state.input.v[2] + deltav[2];

    /* RETURN */
    return(0);
}

/* ENTRY POINT */
int Optimization::ball_post_slave ( Ball* B ) /* RETURN: -- Always
return zero. */
{
    EXECUTIVE* E ;
    E = exec_get_exec();

    /* SEND BALL OBJECT */
    tc_write(&E->monte.work.data_conn,(char*) B, sizeof(Ball));

    /* RETURN */
    return(0);
}
```

Appendix C — *Simulation Code*

SIM_Ball++_L1/M_velocity_target

```
NUM_RUNS: 300

VARS:

ball.obj.state.input.v[0] {km/s} CALC ;
ball.obj.state.input.v[1] {km/s} CALC ;
ball.obj.state.input.v[2] {km/s} CALC ;

DATA:
```

SIM_Ball++_L1/RUN_test/input

```
#include "S_default.dat"
#include "Modified_data/data_record.d"
#include "Modified_data/auto_test.d"

stop = 18000.0 ;

sys.exec.monte.in.activate = Yes ;
sys.exec.monte.in.input_files[0] = "M_velocity_target" ;
sys.exec.sim_com.quiet = Yes ;
```

SIM_Ball++_L1/Modified_data/data_record.d

```
/*
 * Default Data Recording Template.
 */
#ifndef DR_GROUP_ID
#define DR_GROUP_ID sys.exec.record.num_group
#endif

sys.exec.record.group[DR_GROUP_ID].record           = Yes ;
sys.exec.record.group[DR_GROUP_ID].name             = "Ball1" ;
sys.exec.record.group[DR_GROUP_ID].format           = DR_Binary ;
sys.exec.record.group[DR_GROUP_ID].freq             = DR_Always ;
sys.exec.record.group[DR_GROUP_ID].cycle {s}        = 0.1 ;
sys.exec.record.group[DR_GROUP_ID].ref[0] =
    "ball.obj.state.output.r[0][0]" ,
    "ball.obj.state.output.r[0][1]" ,
    "ball.obj.state.output.r[0][2]" ;

DR_GROUP_ID++ ;    /* add 1 to DR_GROUP_ID, THIS SETS DR_GROUP_ID UP
                    * FOR THE NEXT DATA RECORDING FILE */
```

SIM_Ball++_L1/Modified_data/auto_test.d

```
/*
 * Auto Test Data Recording Template.
 */
#ifndef DR_GROUP_ID
#define DR_GROUP_ID sys.exec.record.num_group
#endif
sys.exec.record.group = alloc( DR_GROUP_ID + 1 ) ;

sys.exec.record.group[DR_GROUP_ID].record           = Yes ;
sys.exec.record.group[DR_GROUP_ID].name             = "auto" ;
sys.exec.record.group[DR_GROUP_ID].format           = DR_Fixed_Ascii ;
sys.exec.record.group[DR_GROUP_ID].freq             = DR_Always ;
sys.exec.record.group[DR_GROUP_ID].cycle {s}        = 10.0 ;
sys.exec.record.group[DR_GROUP_ID].ref[0] =
    "ball.obj.state.output.r[0][0]" ,
    "ball.obj.state.output.r[0][1]" ,
    "ball.obj.state.output.r[0][2]" ;

DR_GROUP_ID++ ;    /* add 1 to DR_GROUP_ID, THIS SETS DR_GROUP_ID UP
                    * FOR THE NEXT DATA RECORDING FILE */
```

SIM_Ball++_L1/S_define

```
sim_object { /*=== TRICK EXECUTIVE
========================================*/
  /*=== DATA STRUCTURES ===*/
  sim_services/include: EXECUTIVE exec
(sim_services/include/executive.d) ;

  /*=== JOBS ===*/
  (automatic) sim_services/input_processor:
      input_processor( Inout INPUT_PROCESSOR * IP = &sys.exec.ip ) ;

  (automatic_last) sim_services/exec:
      var_server_sync( Inout EXECUTIVE * E = &sys.exec ) ;

} sys ;
/*============================================================*/

sim_object { /*--- ball ---------------------------------------------
---*/

   /*----- DATA STRUCTURE DECLARATIONS -----*/
             Ball++/L1:  Ball        obj
(Ball++/L1/include/Ball.dd);
   sim_services/include:  INTEGRATOR  integ
(Ball++/L1/include/ball_integ.d);

   /*----- INITIALIZATION JOBS -----*/
   (initialization) Ball++/L1: ball.obj.state_init();

   /*----- EOM DERIVATIVE AND STATE INTEGRATION JOBS -----*/
   (derivative) Ball++/L1: ball.obj.state_deriv();

   (integration) Ball++/L1: ball.obj.state_integ(
      Inout INTEGRATOR * integ = &ball.integ );

} ball; /*----------------------------------------------------------*/

sim_object
{
        /* Data Structure Declarations */
        Ball++/Optim++:  Optimization obj
      (Ball++/Optim++/include/Optimization.dd);


        /* Optimization Initialization and Pre Jobs */
        (monte_master_init)     Ball++/Optim++:
      optimizer.obj.optim_init() ;

        (monte_master_pre)      Ball++/Optim++:
                optimizer.obj.ball_pre_master( Ball* B = &ball.obj ) ;
```

```
        /* Post Optimization Jobs */
        (monte_master_post)    Ball++/Optim++:
      optimizer.obj.ball_post_master() ;

        (monte_slave_post)     Ball++/Optim++:
                optimizer.obj.ball_post_slave( Ball* B = &ball.obj ) ;

} optimizer;

integrate (0.01) ball;
```

Appendix D — *JEOD Targeting Code*

models/Optim++/include/Optimization.dd

```
/************************* TRICK HEADER *****************************
PURPOSE:        (Optimization Default Data)
*********************************************************************/

Optimization.optim_v[0] {m/s} = 0.0 ;
Optimization.optim_v[1] {m/s} = 0.0 ;
Optimization.optim_v[2] {m/s} = 0.0 ;

Optimization.deltav[0] {m/s} = 0.0 ;
Optimization.deltav[1] {m/s} = 0.0 ;
Optimization.deltav[2] {m/s} = 0.0 ;

Optimization.r_des[0]  {m}   = 0.0;
Optimization.r_des[1]  {m}   = 50000000.0;
Optimization.r_des[2]  {m}   = 0.0;

Optimization.pertr[0]  {m}   = 10.0;
Optimization.pertr[1]  {m}   = 10.0;
Optimization.pertr[2]  {m}   = 10.0;
Optimization.pertv[0]  {m/s}   = 0.1;
Optimization.pertv[1]  {m/s}   = 0.1;
Optimization.pertv[2]  {m/s}   = 0.1;
```

models/Optim++/include/Optimization.hh

```
/************************* TRICK HEADER ******************************
PURPOSE:
      ( Header for Optimization class )
LIBRARY DEPENDENCY:
      ((Optimization.o)
       (OptimInit.o)
       (OptimPreJobs.o)
       (OptimPostJobs.o))
*********************************************************************/
#ifndef _OPTIMIZATION_HH_
#define _OPTIMIZATION_HH_
#include "../../dynamics/dyn_body/include/simple_6dof_dyn_body.hh"

class Optimization
{
  public:
   // Constructor and Destructor
   Optimization();
   ~Optimization();

   // Optimization variables
   double optim_v[3];   /* (m/s) optimal initial velocity */
   double deltav[3];  /* (m/s) change in initial velocity*/
   double rf[3]; /* (m) actual final position*/
   double phi[6][6];  /*  state transition matrix*/
   double r_des[3]; /* (m) desired final position*/
   double pertr[3]; /* (m) position perturbations*/
   double pertv[3]; /* (m/s) velocity perturbations*/
   double r0[3]; /*(m) initial position of the first run*/
   double v0[3]; /*(m/s) initial velocity of the first run*/
   double inputs[6]; /*input states to update at each iteration*/
   double outputs[42]; /* store output states */
   int counter;


   // Optimization Initialization
   int optim_init();

   // The "pre" job function. This contains the optimization
   // algorithm and stopping condtions.
   int ball_pre_master(Simple6DofDynBody* S);

   // The "post" job functions. These contain the TCP/IP
   // connection that the master/slave framework uses.
   int ball_post_master();
   int ball_post_slave(Simple6DofDynBody* S);
   int calc_phi();
};
#endif
```

models/Optim++/src/Optimization.cpp

```
/*************************** TRICK HEADER***************************
PURPOSE:
    (Optimization:Optimization optimizer object constructor.)
CLASS:
    (N/A)
LIBRARY DEPENDENCY:
    ((Optimization.o))
PROGRAMMERS:
    ((Kyle Brill, Chun-Yi Wu, Victor Rodriguez)(UT Austin)(May 2010)))
*******************************************************************/

/* System include files. */
#include <iostream>

/* Model include files. */
#include "../include/Optimization.hh"

// Default constructor
Optimization::Optimization() /* RETURN: -- None. */
{
   printf("In Optim constructor.\n");
}

// Destructor
Optimization::~Optimization() /* RETURN: -- None. */
{
   printf("In Optim destructor.\n");
}

/* ENTRY POINT */
int Optimization::calc_phi() /* RETURN: -- Always return zero. */
{
    double perts[6];

    perts[0] = pertr[0];
    perts[1] = pertr[1];
    perts[2] = pertr[2];
    perts[3] = pertv[0];
    perts[4] = pertv[1];
    perts[5] = pertv[2];

    for ( int i = 0 ; i < 6 ; i++ )
    {
        for ( int j = 0 ; j < 6 ; j++ )
        {
                phi[j][i] = ( outputs[j+6*(i+1)] - outputs[j] ) /
perts[i] ;
        }
    }
```

```
     printf("STATE TRANSITION MATRIX\nCalcPHI\n");
     for ( int i = 0 ; i < 6 ; i++ )
     {
         for ( int j = 0 ; j < 6 ; j++ )
         {
            printf("%11.5f  ",phi[i][j]);
         }
         printf("\n");
    }

    /* RETURN */
    return(0);

}
```

models/Optim++/src/OptimInit.cpp

```
/********************************* TRICK HEADER **********************
PURPOSE:
    (Optimization::state_init intializes all necessary variables
CLASS:
    (monte_master_init)
LIBRARY DEPENDENCY:
    ((OptimInit.o))
PROGRAMMERS:
    (((Chun-Yi Wu, Kyle Brill, Victor Rodriguez)(UT Austin)(May 2010))
*********************************************************************/

#include "../include/Optimization.hh"

/* ENTRY POINT */
int Optimization::optim_init() /* RETURN: -- Always return zero. */
{
   /* INITIALIZE ITERATION COUNTER */
   counter = 0;

   /* RETURN */
   return (0) ;
}
```

Optim++/src/OptimPostJobs.cpp

```
/***************************** TRICK HEADER ************************
PURPOSE:
    (Optimization::ball_post_master read slave results via
     TCP/IP Comm and calculate state transition matrix)
LIBRARY DEPENDENCY:
    ((OptimPostJobs.o))
PROGRAMMERS:
    (((Chun-Yi Wu, Kyle Brill, Victor Rodriguez)(UT Austin)(May 2010))
*******************************************************************/

/* System include files. */
#include <iostream>

/* Trick include files. */
#include "sim_services/include/executive.h"
#include "dynamics/dyn_body/include/simple_6dof_dyn_body.hh"
#include "sim_services/include/exec_proto.h"
#include "trick_utils/math/include/trick_math.h"
#include "trick_utils/math/include/vector_macros.h"

/* Model include files. */
#include "../include/Optimization.hh"

/* ENTRY POINT */
int Optimization::ball_post_master() /* RETURN:--Always return zero.*/
{
   printf("\nPOSTMASTER\n");
   double B_mat[3][3], B_inv[3][3], error[3], deltav[3];

   Simple6DofDynBody S_curr ;
   EXECUTIVE* E ;
   E = exec_get_exec() ;

   /* READ SLAVE'S RESULTS */
   printf("\nTC READ\n");
   tc_read( &E->monte.work.data_conn, (char*) &S_curr,
sizeof(Simple6DofDynBody) );

      int modulus = this->counter%7;

      switch(modulus)
      {
      case 0:

   printf("\nCASE 0:\n");
         outputs[0] = S_curr.core_body.state.trans.position[0];
         outputs[1] = S_curr.core_body.state.trans.position[1];
         outputs[2] = S_curr.core_body.state.trans.position[2];
         outputs[3] = S_curr.core_body.state.trans.velocity[0];
         outputs[4] = S_curr.core_body.state.trans.velocity[1];
```

```
         outputs[5] = S_curr.core_body.state.trans.velocity[2];
   printf("OUTPUT FROM CORE BODY\nrx = %f\n ry = %f\n rz = %f\n vx =
%f\n vy = %f\n vz =
%f\n\n",outputs[0],outputs[1],outputs[2],outputs[3],outputs[4],outputs
[5]);
         this->counter++;
   printf("\nBREAK\n");
         break;
      case 1:

   printf("\nCASE 1:\n");
         outputs[6] = S_curr.core_body.state.trans.position[0];
         outputs[7] = S_curr.core_body.state.trans.position[1];
         outputs[8] = S_curr.core_body.state.trans.position[2];
         outputs[9] = S_curr.core_body.state.trans.velocity[0];
         outputs[10] = S_curr.core_body.state.trans.velocity[1];
         outputs[11] = S_curr.core_body.state.trans.velocity[2];
         this->counter++;
         break;
      case 2:

   printf("\nCASE 2:\n");
         outputs[12] = S_curr.core_body.state.trans.position[0];
         outputs[13] = S_curr.core_body.state.trans.position[1];
         outputs[14] = S_curr.core_body.state.trans.position[2];
         outputs[15] = S_curr.core_body.state.trans.velocity[0];
         outputs[16] = S_curr.core_body.state.trans.velocity[1];
         outputs[17] = S_curr.core_body.state.trans.velocity[2];
         this->counter++;
         break;
      case 3:

   printf("\nCASE 3:\n");
         outputs[18] = S_curr.core_body.state.trans.position[0];
         outputs[19] = S_curr.core_body.state.trans.position[1];
         outputs[20] = S_curr.core_body.state.trans.position[2];
         outputs[21] = S_curr.core_body.state.trans.velocity[0];
         outputs[22] = S_curr.core_body.state.trans.velocity[1];
         outputs[23] = S_curr.core_body.state.trans.velocity[2];
         this->counter++;
         break;


      case 4:

   printf("\nCASE 4:\n");
         outputs[24] = S_curr.core_body.state.trans.position[0];
         outputs[25] = S_curr.core_body.state.trans.position[1];
         outputs[26] = S_curr.core_body.state.trans.position[2];
         outputs[27] = S_curr.core_body.state.trans.velocity[0];
         outputs[28] = S_curr.core_body.state.trans.velocity[1];
         outputs[29] = S_curr.core_body.state.trans.velocity[2];
```

```
      this->counter++;
      break;
   case 5:

printf("\nCASE 5:\n");
      outputs[30] = S_curr.core_body.state.trans.position[0];
      outputs[31] = S_curr.core_body.state.trans.position[1];
      outputs[32] = S_curr.core_body.state.trans.position[2];
      outputs[33] = S_curr.core_body.state.trans.velocity[0];
      outputs[34] = S_curr.core_body.state.trans.velocity[1];
      outputs[35] = S_curr.core_body.state.trans.velocity[2];
      this->counter++;
      break;
   case 6:

printf("\nCASE 6:\n");
      outputs[36] = S_curr.core_body.state.trans.position[0];
      outputs[37] = S_curr.core_body.state.trans.position[1];
      outputs[38] = S_curr.core_body.state.trans.position[2];
      outputs[39] = S_curr.core_body.state.trans.velocity[0];
      outputs[40] = S_curr.core_body.state.trans.velocity[1];
      outputs[41] = S_curr.core_body.state.trans.velocity[2];

      calc_phi();
      printf("STATE TRANSITION MATRIX\nPostmaster\n");

      for ( int i = 0 ; i < 6 ; i++ )
      {
         for ( int j = 0 ; j < 6 ; j++ )
         {
            printf("%11.5f  ",phi[i][j]);
         }
         printf("\n");
      }

      for(int i = 0; i <3; i++){
              for(int j = 0; j<3; j++){
                      B_mat[i][j] = this->phi[i][j+3];
              }
      }


      rf[0] = outputs[0];
      rf[1] = outputs[1];
      rf[2] = outputs[2];

      V_SUB(error, this->r_des, this->rf);
      dm_invert(B_inv, B_mat);
      MxV(deltav, B_inv, error);

      inputs[3] += deltav[0];
      inputs[4] += deltav[1];
```

```
         inputs[5] += deltav[2];

         this->optim_v[0] = inputs[3];
         this->optim_v[1] = inputs[4];
         this->optim_v[2] = inputs[5];

         printf("\nPOST/rf:\n");
         V_PRINT(rf);

         this->counter++;
         printf("\nTarget #:    %d\n", this->counter/7 );
         printf("rx = %f\t", outputs[0]);
         printf("ry = %f\t", outputs[1]);
         printf("rz = %f\t", outputs[2]);
         printf("vx = %f\t", outputs[3]);
         printf("vy = %f\t", outputs[4]);
         printf("vz = %f\n", outputs[5]);

         break;

       }
   /* RETURN */
   printf("\nRETURN\n");
   return(0);
}

/* ENTRY POINT */
int Optimization::ball_post_slave ( Simple6DofDynBody* S )
{
   EXECUTIVE* E ;
   E = exec_get_exec();

   printf("\nPOSTSLAVE\n");
   /* Send F(x) - which is in BSTATE */
   tc_write(&E->monte.work.data_conn,(char*) S,
sizeof(Simple6DofDynBody));

   return(0);

}
```

Optim++/src/OptimPreJobs.cpp

```
/************************** TRICK HEADER ***************************
PURPOSE:
    (Optimization::ball_pre_master contains the targeting algorithm
     and end condition)
CLASS:
    (monte_master_pre)
LIBRARY DEPENDENCY:
    ((OptimPreJobs.o))
PROGRAMMERS:
    (((Chun-Yi Wu, Kyle Brill, Victor Rodriguez)(UT Austin)(May 2010))
*******************************************************************/

/* System include files. */
#include <iostream>
/* Trick include files. */
#include "sim_services/include/exec_proto.h"
#include "trick_utils/math/include/trick_math.h"
#include "trick_utils/math/include/vector_macros.h"

/* Model include files. */
#include "../include/Optimization.hh"
#include "../../dynamics/dyn_body/include/simple_6dof_dyn_body.hh"
#include
"../../dynamics/body_action/include/dyn_body_init_trans_state.hh"

/* ENTRY POINT */
int Optimization::ball_pre_master(Simple6DofDynBody* S) /* RETURN: --
Always return zero. */
{
   /* INCREMENT ITERATION COUNTER */
   printf("\nPREMASTER\n");
   printf("\nTARGETING THIS POSITION: \nr_x = %f m\nr_y = %f m\nr_z =
%f m\n",r_des[0],r_des[1],r_des[2]);
   double error[3];

   if ( this->counter > 0 )
   {
      /* CALCULATE THE DIFFERENCE BETWEEN THE DESIRED FINAL POSITION
         AND THE ACTUAL FINAL POSITION */
      V_SUB( error , this->r_des , this->rf );

  /* ABSOLUTE VALUE OF THE MAXIMUM ERROR OF ALL POSITION COMPONENTS */
      double max = error[0];
      if ( -error[1] > max )
         max = -error[1];
      if ( error[1] > max )
         max = error[1];
      if ( -error[0] > max )
         max = -error[0];
      if ( error[2] > max )
```

```
         max = error[2];
      if ( -error[2] > max )
         max = -error[2];

      printf("\nIteration #: %d\n", this->counter);
      printf("\nFinal Position:\nr_x = %f km\nr_y = %f km\nr_z = %f
km\n",rf[0],rf[1],rf[2]);

/* STOPPING CONDITION IS MAX ERROR < .00001 km OR 20 ITERATIONS */
      if( max < 0.00001 || this->counter == 30)
      {
         printf("\n*******************\n");
         printf("The change in velocity necessary to reach the desired
final location\n\n");
         printf("r_des_x = %f km\nr_des_y = %f km\nr_des_z = %f
km\n",r_des[0],r_des[1],r_des[2]);
         printf("\nfrom the initial state\n\nr_x0 = %f km\nr_y0 = %f
km\nr_z0 = %f km\nv_x0 = %f km/s\nv_y0 = %f km/s\nv_z0 = %f
km/s\n",r0[0],r0[1],r0[2],v0[0],v0[1],v0[2]);
         printf("\nis\n\ndeltav_x = %f km/s\ndeltav_y = %f
km/s\ndeltav_z = %f km/s\n",optim_v[0]-v0[0],optim_v[1]-
v0[1],optim_v[2]-v0[2]);
         printf("\nThe targeter took %d iterations\n",this-
>counter/7);
         printf("*******************\n\n\n");

      /* TERMINATES THIS FUNCTION ONCE THE END CONDITION IS REACHED */
         exec_terminate("ball_pre_master","End Condition reached.");
      }
      else
      {
         printf("\nContinuing algorithm...end condition not
met.\n\n");
      }

      int modulus = this->counter%7;
      double temp;

      switch(modulus)
      {
      case 0:

         S->core_body.state.trans.velocity[0] = this->optim_v[0] ;
         S->core_body.state.trans.velocity[1] = this->optim_v[1] ;
         S->core_body.state.trans.velocity[2] = this->optim_v[2] ;
         break;

     case 1:

         temp = inputs[0] + pertr[0];
         S->core_body.state.trans.position[0] = temp;
         break;
```

```
      case 2:

         temp = inputs[1] + pertr[1];
         S->core_body.state.trans.position[1] = temp;
         break;
      case 3:

         temp = inputs[2] + pertr[2];
         S->core_body.state.trans.position[2] = temp;
         break;
      case 4:

         temp = inputs[3] + pertv[0];
       S->core_body.state.trans.velocity[0] = temp;
         break;
      case 5:

         temp = inputs[4] + pertv[1];
        S->core_body.state.trans.velocity[1] = temp;
         break;
      case 6:

         temp = inputs[5] + pertv[2];
         S->core_body.state.trans.velocity[2] = temp;
         break;

      }

   }
   else
   {
      /* STORE FIRST INTIAL VELOCITY SO WE CAN SUGGEST A DELTA V
         AT THE END OF THE ALGORITHM */
      this->r0[0] = S->core_body.state.trans.position[0];
      this->r0[1] = S->core_body.state.trans.position[1];
      this->r0[2] = S->core_body.state.trans.position[2];
      this->v0[0] = S->core_body.state.trans.velocity[0];
      this->v0[1] = S->core_body.state.trans.velocity[1];
      this->v0[2] = S->core_body.state.trans.velocity[2];


      printf("Using intial conditions\nrx = %f\n ry = %f\n rz = %f\n
vx = %f\n vy = %f\n vz = %f\nfor the first reference
trajectory\n",r0[0],r0[1],r0[2],v0[0],v0[1],v0[2]);
//      printf("Using intial conditions\nrx = %f\n ry = %f\n rz = %f\n
vx = %f\n vy = %f\n vz = %f\nfor the first reference trajectory\n",T-
>position[0],T->position[1],T->position[2],T->velocity[0],T-
>velocity[1],T->velocity[2]);

      inputs[0] = S->core_body.state.trans.position[0];
      inputs[1] = S->core_body.state.trans.position[1];
      inputs[2] = S->core_body.state.trans.position[2];
```

```
      inputs[3] = S->core_body.state.trans.velocity[0];
      inputs[4] = S->core_body.state.trans.velocity[1];
      inputs[5] = S->core_body.state.trans.velocity[2];

      /* RETURN -- this is the first run, nothing else left to do*/
      return (0);
   }

   /* RETURN */
   return (0);

}
```

SIM_1_Optim/Modified_data/Earth/grav_controls.d

```
/*
PURPOSE:
   (This data file sets up the vehicle gravity model controls.)

REFERENCE:
   ((JSC Engineering Orbital Dynamics Models))

ASSUMPTIONS AND LIMITATIONS:
   ((?))

PROGRAMMERS:
   (((Edwin Z. Crues) (NASA) (November 2008) (--) (JEOD 2.0 Testing)))
*/

#define GC_EARTH 0

/* Associate the gravity controls for the DynBody. */
VEH_OBJ.body.grav_interaction.grav_controls = VEH_OBJ.grav_controls;
VEH_OBJ.body.grav_interaction.n_grav_controls = 1;

/* Set up the gravity controls for the Earth. */
VEH_OBJ.grav_controls[GC_EARTH].planet_name = "Earth";
VEH_OBJ.grav_controls[GC_EARTH].active      = True;
VEH_OBJ.grav_controls[GC_EARTH].spherical   = True;
VEH_OBJ.grav_controls[GC_EARTH].degree      = 0;
VEH_OBJ.grav_controls[GC_EARTH].order       = 0;
```

SIM_1_Optim/Modifed_data/Integrator/integrator.d

```
// Set the integration options.
dynamics.integ.option = Runge_Kutta_4;
dynamics.integ.first_step_deriv = True;
```

SIM_1_Optim/Modified_data/vehicle/veh_state.d

```
// State initialization data for a typical ISS orbital state.

//
// Set the translational position.
//
VEH_OBJ.trans_init.subject = &VEH_OBJ.body;
VEH_OBJ.trans_init.reference_ref_frame_name = "Earth.inertial";
VEH_OBJ.trans_init.body_frame_id     = "composite_body";
VEH_OBJ.trans_init.position[0] {M}   = 42000000.0, 0.0, 0.0;
VEH_OBJ.trans_init.velocity[0] {M/s} = 0.0, 3500, 0.0;

//
// Set the rotational position.
//
VEH_OBJ.lvlh_init.subject                            = &VEH_OBJ.body;
VEH_OBJ.lvlh_init.planet_name                        = "Earth";
VEH_OBJ.lvlh_init.body_frame_id                      = "composite_body";
VEH_OBJ.lvlh_init.orientation.data_source    =
Orientation::InputEulerRotation;
VEH_OBJ.lvlh_init.orientation.euler_sequence      = Yaw_Pitch_Roll;
VEH_OBJ.lvlh_init.orientation.euler_angles[0] {d} = 1.0, 85.0, 0.0;
VEH_OBJ.lvlh_init.ang_velocity[0]           {d/s} = 0.0, 0.0, 0.0;
```

SIM_1_Optim/S_define

```
/*********************************************************************
 *           JSC Engineering Orbital Dynamics Tutorial Sim_1         *
 *-------------------------------------------------------------------*
 * This is the simulation definition file for the  JSC Engineering
Orbital   *
 * Dynamics for tutorial sims. It represents an example S_define for a
*
 * the simulation of a single 6 degree of freedom object , a spinning
stick  *

********************************************************************/
/*********************************************************************
 *       Author: A.A. Jackson
 *         Date: March 2009
 *       E-Mail: albert.a.jackson@nasa.gov
 *        Phone: 281-483-5037
 * Organization: ESCG, Mail Code JE07
 *               Simulation & Graphics Branch
 *               Software, Robotics & Simulation Division
 *               2101 NASA Parkway
 *               Houston, Texas 77058
 *-------------------------------------------------------------------*
 * Modified By: Christopther Thebeau
 *        Date: August 2009
 * Description: Cleaned up to make all sims consistant
 *********************************************************************
//
//          sys - Trick runtime executive and data recording routines
//         time - Universal time
//          env - Environment: gravity
//     earth - Planet environment model
//       sv_dyn - Space vehicle dynamics model
//     dynamics - Orbital dynamics
//
//===================================================================


// Define job calling intervals
#define LOW_RATE_ENV  60.00    // Low-rate environment update interval
#define DYNAMICS       0.03125 // Vehicle and plantary dynamics
interval (32Hz)


// Define the phase initialization priorities.
// NOTE: Initialization jobs lacking an assigned phase initialization
priority
// run after all initialization jobs that have assigned phase init
priorities.
#define P_TIME  P10   // Highest priority; these jobs depend on time
```

```
#define P_MNGR  P20   // Dynamics manager initializations
#define P_ENV   P30   // Environment initializations
#define P_BODY  P40   // Orbital body initializations
#define P_DYN   P50   // State-dependent initializations



//=====================================================================
// SIM_OBJECT: sys
// This is the Trick executive model; this model should be basically
// the same for all Trick applications.
//=====================================================================
sim_object {

   // Data structures
   sim_services/include: EXECUTIVE exec
(sim_services/include/executive.d);

   // Automatic jobs
   sim_services/input_processor: input_processor (
      Inout INPUT_PROCESSOR *IP = &sys.exec.ip);

} sys;

//=====================================================================
// SIM_OBJECT: time
// This sim object relates simulation time to time on the Earth.
//=====================================================================

sim_object {
   // Data structures
   environment/time: TimeManager manager;
   environment/time: TimeManagerInit manager_init;

   // Time Scales
   environment/time: TimeTAI tai;
   environment/time: TimeUTC utc;

   // Time Converters
   environment/time: TimeConverter_Dyn_TAI  conv_dyn_tai;
   environment/time: TimeConverter_TAI_UTC  conv_tai_utc
       (environment/time/data/tai_to_utc.d);

   // Initialization jobs
   // Register the basic time scales with the time manager.

   // TAI
   P_TIME (initialization) environment/time:
   time.manager.register_type(
      In     Time      & time_reg = time.tai);
   P_TIME (initialization) environment/time:
   time.manager.register_converter(
```

```
      In      TimeConverter   & time_conv = time.conv_dyn_tai);

   // UTC
   P_TIME (initialization) environment/time:
   time.manager.register_type(
     In Time          & time_reg = time.utc );
   P_TIME (initialization) environment/time:
   time.manager.register_converter(
     In TimeConverter & time_conv = time.conv_tai_utc );

   // Initialize the time manager.
   P_TIME (initialization)  environment/time:
   time.manager.initialize(
      Inout  TimeManagerInit * manager_init = &time.manager_init);

   // Compute appropriate calendar dates at initialization.
   P_TIME (initialization) environment/time: time.utc.calendar_update(
      In double simtime = sys.exec.out.time );

   // Scheduled jobs
   // Update Time Scales
   (DYNAMICS, environment) environment/time:
   time.manager.update(
      In    double    simtime =  sys.exec.out.time);

   // Update the calendar times of interest.
   (DYNAMICS, environment) environment/time: time.utc.calendar_update(
      In double simtime = sys.exec.out.time );

} time;


//=====================================================================
// SIM_OBJECT: env
// This sim object models the space environment.
//=====================================================================
sim_object {

   // Data structures
   environment/gravity:      GravityModel          gravity;

   // Initialization
   P_ENV (initialization) environment/gravity:
   env.gravity.initialize_model (
      Inout DynManager & dyn_manager = dynamics.manager );


} env;


//=====================================================================
// SIM_OBJECT: earth
```

```
// This sim object models the space environment.
//======================================================================
sim_object {

   // Data structures
   environment/planet:  Planet               planet
      (environment/planet/data/earth.d);

   environment/gravity:    GravityBody          gravity_body;

   environment/gravity:    GravityCoeffs        gravity_coefs
      (environment/gravity/data/earth_GGM02C.d);

   // Initialization jobs
   P_ENV (initialization) environment/gravity:
   earth.gravity_body.initialize_coefs(
      In GravityCoeffs & coefs = earth.gravity_coefs );

   P_ENV (initialization) environment/gravity:
   env.gravity.add_grav_body(
      Inout GravityBody & grav_body = earth.gravity_body );

   P_ENV (initialization) environment/planet:
   earth.planet.register_model(
      Inout GravityBody & grav_body   = earth.gravity_body,
      Inout DynManager  & dyn_manager = dynamics.manager );

   P_BODY (initialization) environment/planet:
   earth.planet.initialize( );

} earth;

//======================================================================
// SIM_OBJECT: sv_dyn
// This sim object models a vehicle in space.
//======================================================================
sim_object {

   // Data structures
   // Dynamical propagation and initial state.
   dynamics/dyn_body:    Simple6DofDynBody       body;
   dynamics/body_action: DynBodyInitTransState   trans_init;
   dynamics/body_action: DynBodyInitRotState     rot_init;
   dynamics/body_action: DynBodyInitLvlhRotState lvlh_init;

   // Vehicle mass initialization parameters.
   dynamics/body_action: MassBodyInit mass_init;

   // Vehicle derived reference frames.
   dynamics/derived_state: EulerDerivedState     euler;
   dynamics/derived_state: PlanetaryDerivedState pfix;
   dynamics/derived_state: LvlhDerivedState      lvlh;
```

```
   dynamics/derived_state: EulerDerivedState     lvlh_euler;
   dynamics/derived_state: OrbElemDerivedState   orb_elem;

   // Vehicle perturbation forces and torques.
   dynamics/dyn_body: Force  force_extern;
   dynamics/dyn_body: Torque torque_extern;

   // Vehicle environmental forces and torques.
   environment/gravity:      GravityControls  grav_controls[1];

   // Initialization jobs
   P_ENV (initialization) dynamics/dyn_body:
   sv_dyn.body.initialize_model(
      Inout DynManager & manager = dynamics.manager );

   P_DYN (initialization) dynamics/derived_state:
   sv_dyn.euler.initialize(
      Inout DynBody    & subject_body = sv_dyn.body,
      Inout DynManager & dyn_manager  = dynamics.manager );

   P_DYN (initialization) dynamics/derived_state:
   sv_dyn.pfix.initialize(
      Inout DynBody    & subject_body = sv_dyn.body,
      Inout DynManager & dyn_manager  = dynamics.manager );

   P_DYN (initialization) dynamics/derived_state:
   sv_dyn.lvlh.initialize(
      Inout DynBody    & subject_body = sv_dyn.body,
      Inout DynManager & dyn_manager  = dynamics.manager );

   P_DYN (initialization) dynamics/derived_state:
   sv_dyn.lvlh_euler.initialize(
      In    RefFrame   & ref_frame    = sv_dyn.lvlh.lvlh_frame,
      Inout DynBody    & subject_body = sv_dyn.body,
      Inout DynManager & dyn_manager  = dynamics.manager );

   P_DYN (initialization) dynamics/derived_state:
   sv_dyn.orb_elem.initialize(
      Inout DynBody    & subject_body = sv_dyn.body,
      Inout DynManager & dyn_manager  = dynamics.manager );

   (initialization) dynamics/derived_state: sv_dyn.euler.update( );

   (initialization) dynamics/derived_state: sv_dyn.pfix.update( );

   (initialization) dynamics/derived_state: sv_dyn.lvlh.update( );

   (initialization) dynamics/derived_state: sv_dyn.lvlh_euler.update(
);

   (initialization) dynamics/derived_state: sv_dyn.orb_elem.update( );
```

```
   // Environment class jobs
   (DYNAMICS, environment) dynamics/derived_state:
   sv_dyn.euler.update( );

   (DYNAMICS, environment) dynamics/derived_state:
   sv_dyn.pfix.update( );

   (DYNAMICS, environment) dynamics/derived_state:
   sv_dyn.lvlh.update( );

   (DYNAMICS, environment) dynamics/derived_state:
   sv_dyn.lvlh_euler.update( );

   (DYNAMICS, environment) dynamics/derived_state:
   sv_dyn.orb_elem.update( );

} sv_dyn;
```

```
sim_object
{
        /* Data Structure Declarations */
        Optim++:  Optimization   obj
(Optim++/include/Optimization.dd);


        /* Optimization Initialization and Pre Jobs */
        (monte_master_init)     Optim++:
optimizer.obj.optim_init() ;

        (monte_master_pre)      Optim++:
                optimizer.obj.ball_pre_master(
Simple6DofDynBody* S = &sv_dyn.body);
/*,   DynBodyInitTransState* T = &sv_dyn.trans_init);*/

        /* Post Optimization Jobs */
        (monte_master_post)    Optim++:
optimizer.obj.ball_post_master() ;

        (monte_slave_post)     Optim++:
                optimizer.obj.ball_post_slave(
Simple6DofDynBody* S = &sv_dyn.body ) ;

} optimizer;
```

```
/********************************************************************
 COLLECT: Vehicle force and torque collection statements for dynamics
*********************************************************************/
//====================
// Force collections
//====================
// Collect effector forces for vehicle or forces from outside of jeod
vcollect sv_dyn.body.collect.collect_effector_forc
CollectForce::create {
  sv_dyn.force_extern
};

// Collect dynamic environmental forces for vehicle
vcollect sv_dyn.body.collect.collect_environ_forc CollectForce::create
{
};

//=====================
// Torque collections
//=====================
// Collect effector torques for vehicle or torques from outside of
jeod
vcollect sv_dyn.body.collect.collect_effector_torq
CollectTorque::create {
  sv_dyn.torque_extern
};

// Collect dynamic environmental torques for vehicle
vcollect sv_dyn.body.collect.collect_environ_torq
CollectTorque::create {
};

//====================================================================
// SIM_OBJECT: dynamics
// This sim object manages the key dynamic elements of the simulation.
//====================================================================
sim_object {

   // Data structures
   dynamics/dyn_manager: DynManager       manager;
   dynamics/dyn_manager: DynManagerInit   manager_init;
   dynamics/body_action: BodyAction     * body_action_ptr;
   sim_services/include: INTEGRATOR       integ;
   utils/message:        TrickMessageHandler   msg_handler;

   // Jobs registered for input file activation.
   (0.0, environment) dynamics/dyn_manager:
   dynamics.manager.add_body_action(
      Inout BodyAction * body_action = dynamics.body_action_ptr );

   // Initialization jobs
   P_MNGR (initialization) dynamics/dyn_manager:
```

```
   dynamics.manager.initialize_model(
      Inout INTEGRATOR *     integ = &dynamics.integ,
      In    DynManagerInit & init  = dynamics.manager_init );

   P_BODY (initialization) dynamics/dyn_manager:
   dynamics.manager.initialize_simulation( );

   // Derivative class jobs
   P_MNGR Idynamics (derivative) dynamics/dyn_manager:
   dynamics.manager.gravitation( );

   (derivative) dynamics/dyn_manager:
   dynamics.manager.compute_derivatives( );

   // Integration jobs
   (integration) dynamics/dyn_manager:
   dynamics.manager.integrate (
      Inout INTEGRATOR  * integ     = &dynamics.integ,
      In    double        sim_time0 =  sys.exec.out.time,
      Inout TimeManager & time_mgr  =  time.manager );

} dynamics;

integrate (DYNAMICS) dynamics;
```