Optimization in Trick

Prepared by:

Sean Moles
Zach Tschirhart
Brendan Tseung

Table of Contents

1.0 Introduction 2
1.1 Purpose 2
1.2 Background 2
2.0 Plan of Action 3
2.1 Establishing a Goal 3
2.2 Investigate Documentation 3
2.3 Implement Solution in Trick using C and C++ 3
3.0 Optimization 4
3.1 Description of Method 4
3.2 Rationale 5
3.3 Document Discrepancies Encountered 5
4.0 C Implementation 6
4.1 Adding the Jet 6
4.2 Creating the Optimization Framework 7
4.3 The Simulation Files 10
4.4 Running the Simulation 12
5.0 C++ Implementation 13
5.1 The Jet in C++ 13
5.2 Optimization in C++ 14
5.3 Simulation Files with C++ models 16
5.4 Running the Simulation 17
5.4 Running the Simulation 18
6.0 Future Work 21
7.0 Reference 22
8.0 Appendix 23
8.1 Modified C model code 23
8.2 Simulation Files needed for the ball and optim models (in C) 29
8.3 Modified C++ model code 31
8.4 Simulation files needed for the Ball++ and Optim++ models (in C++) 38

1.0 Introduction

This project was designed to solve the issue of implementing an iterative optimization algorithm in Trick. The goal was to first define a concrete, achievable objective around which a plan of action could be developed, and then implement an optimization method in Trick. Once an optimization method had been determined, the next step was to implement it in a simple proof-of-concept Trick simulation so that eventually we have the necessary tools to eventually integrate the targeting algorithm in a Trick simulation using JEOD.

1.1 Purpose

The purpose of this project was to create an iterative method framework in Trick in an effort to optimize a particular solution. On the basis of ease-of-use and reusability, the objective was refined to state that our optimization method *would be implemented in a way that takes advantage of Trick features to the greatest possible extent and facilitates easy modification in the future.*

1.2 Background

This information contains C, C++, and Trick coding. Future implementations will possibly include MATLAB and JEOD code. There is also a need for understanding Monte Carlo and the Master/Slave set-up in Trick which can be found in the Trick Tutorials Documentation.

2.0 Plan of Action

2.1 Establishing a goal

The first step taken in the project was to define a concrete, achievable objective around which a plan of action could be developed. The goal of the project was to implement an optimization method in Trick, however, from the start of the project it was clear that there were numerous methods for accomplishing this. As stated before, we decided to refine our objective to state that our solution to the problem *would be implemented in a way that takes advantage of Trick features to the greatest possible extent and facilitates easy modification in the future.*

2.2 Investigate documentation

The next goal was to investigate the documentation and tutorials included with Trick. There was the obvious possibility of developing a solution that worked independent of the Trick framework but this was less desirable since it was in direct opposition to the stated objective.

2.3 Implement solution in Trick using C and C++

Once an optimization method had been determined, the next step was to implement it in a simple proof-of-concept Trick simulation. Because the team had extensive experience using Trick with C, but also planned on eventually using C++ when working later with JEOD, the method would first be implemented relatively quickly in C. Once it had been proven, it could be implemented in C++, without dealing with an unexpected problems that might arise from first developing a program utilizing both the Trick optimization structure and the C++ framework given the team's relative inexperience with both. For simply proving that the method chosen was feasible, the Trick simulation developed would be a straightforward modification of the ball example used throughout the Trick documentation.

3.0 Optimization Framework and Monte Carlo method

3.1 Description of method

Based on the information gleaned from the Trick User's Guide, the best method that we found for implementing an iterative targeting program within Trick was to use the optimization framework built into the Trick software. The optimization framework is essentially an adaptation of Trick's Monte Carlo method, which allows the user to "run the simulation repeatedly over a varying input space" [1]. With the optimization framework, the simulation inputs can be modified between each run based on the results of previous simulation runs, rather than using semi-random inputs as in the case of a Monte Carlo simulation.

Optimization in Trick uses a master/slave system, where the master program defines the simulation parameters and sends them to the slave programs, which run an iteration of a standard Trick simulation and then return the results to the master program, which then recalculates the simulation inputs based on the results. While it probably won't be useful for the orbital targeting program (the focus of this project), it could potentially be used to easily adapt an optimization program to run on a distributed computing system if the optimization algorithm was the sort that would benefit from running multiple simulations for each iteration.

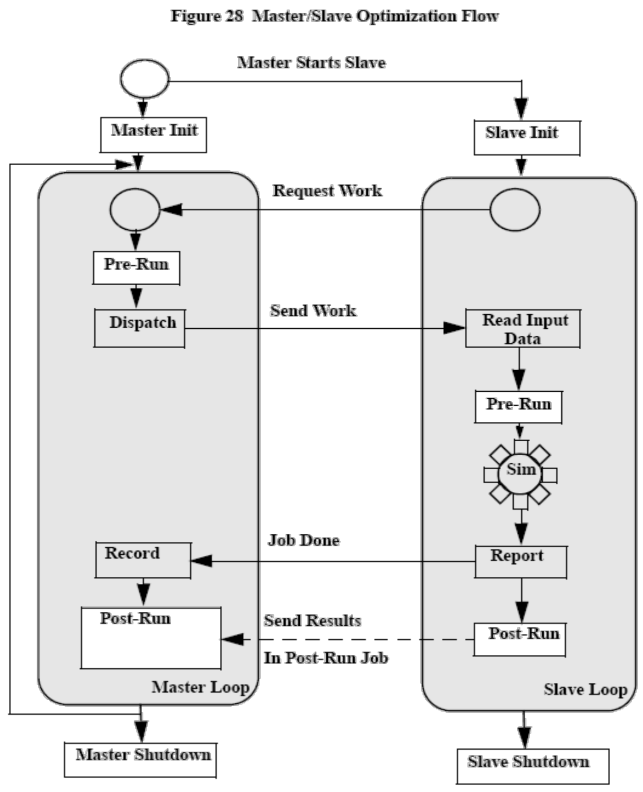

Figure 1 The master/slave structure used by Trick's optimization method [1]

3.2 Rationale

Due to the numerous methods available, a modified Monte Carlo algorithm method was chosen for its ease of adaptation, which allowed us to use the examples from the Trick tutorial guide and view how to implement this in C. The C++ version was just a modified version from the C code. Having both the C and C++ versions allowed for more compatibility and a object-oriented piece of code, which is very important in making code more usable for future students. Our next step would be to use this idea and apply it to an existing JEOD simulation and add on to that design.

This method was also chosen because it is described entirely within the Trick documentation, which reduces the probability of encountering unexpected problems if the work done in this project is ever reused in future programs.

3.3 Documentation Discrepancies Encountered

Section 17.4 of the Trick User's Guide, "Activating Monte Carlo", provides the following code to set the simulation variables required by both the Monte Carlo and optimization methods[2]:

```
sys.exec.monte.in.activate = Yes ;
sys.exec.monte.in.input_file = "<Monte Carlo input file>" ;
```

The version of Trick in use in this project (07.18.1) doesn't recognize the second input variable when the simulation is run. Further experimentation demonstrated that the correct input variable is:

```
sys.exec.monte.in.input_files[0] = "<Monte Carlo input
file>" ;
```

The documentation may not actually match the Trick version 07.18.1. It gives a documentation number of 2005.7.0 which appears to be a different numbering system.

4.0 C Implementation

The L1 ball simulation (a ball acted on by a constant point force) is given as examples in Trick as both C and C++ simulations, which was one of the reasons why we decided to implement the optimization framework with this particular simulation. Although the base simulation was already in place, we still needed to create two things:

- A variable that could be easily manipulated and optimized
- An optimization framework

The Trick Tutorial already has both of these examples in C but with the cannonball simulation. In Section 10.0, the tutorial shows how to add propulsion to the cannonball and in Section 13.0, it shows how to optimize the time the jets fire with the amoeba algorithm in order to reach the maximum distance for a cannonball1. We hope to demonstrate both of these features in a more simplified setting with the L1 ball simulation.

4.1 Adding the Jet

4.1.1 Modifying the Header files and the Data files

We started by making sure that all of our variables were in place for the jet to be used. In *ball_state.h*, we added five jet variables to the BSTATE_OUT data structure:

```
/* Variables needed for Jet addition */
int jet_on ;                  /* -- 0|1 Jet firing? */
int jet_count ;               /* -- How many jet firings? */
double force_jet[3] ;         /* N force of jet */
double force_jet_plus ;       /* N Configurable force of jet in +X direction */
double time_to_fire_jet ;     /* s jet fire time */
```

Some of the variables should also have initial values, so we also added lines to the *ball_state.d* file:

```
/* Initial data needed for jet */
BSTATE.out.force_jet_plus  {N} = 5.0 ;
BSTATE.out.time_to_fire_jet {s} = 0.0 ;
```

4.1.2 Adding the functions for the jets

The jet functions created for the project are very similar to the ones that are shown in Section 10.0 of the Trick tutorials with a few minor modifications [2]. One of the more important ones that we have to notice is that we decided to have the jet fire along the x-axis instead of the z-axis since the ball simulation is a 2D simulation.

The *ball_jet_control.c* function tells the simulation when to turn on the jet.

```
/************************ TRICK HEADER ******************************
PURPOSE:                ( Controls when jets are fired )
********************************************************************/

#include "../include/ball_state.h"
#include "sim_services/include/exec_proto.h"
#include "trick_utils/math/include/trick_math.h"

/* Define what equality is */
#define BALL_EQUALS(X,Y) ( fabs(X-Y) < 1.0e-9 ) ? 1 : 0

int ball_jet_control( BSTATE* B )
{
        /* Declare varaible for simulation time */
        double sim_time ;
        sim_time = exec_get_sim_time() ;

        /* If the simulation time equals the time that the jet fires, then
           turn the jet on */
        if(BALL_EQUALS(sim_time, roundoff(.1, B->out.time_to_fire_jet)))
        {
                B->out.jet_on = 1 ;
        }
        return(0) ;
}
```

The *ball_force_jet.c* function activates the jet whenever the jet is declared as on.

```
/******************************* TRICK HEADER **********************
PURPOSE:                ( Jet fire force )
******************************************************************/

#include "../include/ball_state.h"

int ball_force_jet( BSTATE *B )
{
        if ( B->out.jet_on )
        {
                /* If jet is on, then assign the jet a certain force
                   and then turn the jet off. */
                B->out.force_jet[0] = B->out.force_jet_plus ;
                B->out.jet_on = 0 ;
        }
        else
                B->out.force_jet[0] = 0.0 ;

        return ( 0 );
}
```

4.2 Creating the Optimization Framework

In Section 13.0 of the Trick Tutorial, an amoeba algorithm was presented [2]. However, in our build of Trick, that code was missing, so it was important for us to build this optimization framework from the ground up around the L1 ball simulation so that we could eventually transfer it to the L1 ball simulation that's written in C++. Because we wanted to keep our project simple so that we could focus on the problem, the "algorithm" that we decided to implement wasn't really an algorithm, but just an increment to see if the framework was truly working and that an actual algorithm could eventually take its place in the future.

We decided to create a separate model directory (trick_models/optim) to place all of our new work into one place. In retrospect, it might have been better to place it inside the ball model directory (trick_models/ball) because it optimizes certain variables from that simulation. However, it is important to understand also that this framework can still be applied to any simulation.

4.2.1 Optimization Data structures

For our project, we tried to keep things simple, so the only necessary variable that we needed in our data structure was the variable that we would be optimizing on the algorithm side. In most cases, this optimization data structure would contain variables that are needed for the optimization algorithm, but since we're doing a simple increment (just to show that each subsequent optimization is refined through each run), it wasn't necessary for us to create a complex algorithm.

Our optimization data structure:

```
typedef struct {

        double time;    /* s JET_TIME */

} JET_TIME ;
```

4.2.2 Optimization and the Master/Slave Framework

For any iterative algorithm it is always important to initialize beginning values, so we included a place holder, *optim_init.c*, inside our model, but in reality does not serve any purpose unless we actually decide to use a real algorithm.

The heart and soul of the optimization framework actually lies with the pre-master job because this function actually contains the optimization algorithm. Ending conditions, which are demonstrated, can be implemented while using this framework, which is one of the key goals of this project.

```
/********************************* TRICK HEADER ************************************
PURPOSE:                (Pre master algorithm to modify ball position)
*************************************************************************************/

#include "ball/L1/include/ball_state.h"
#include "../include/jet_time.h"

int ball_pre_master(BSTATE* B,
                    JET_TIME* JT)
{
        /* "Algorithm" that increments the optimal time by 1 */
        JT->time += 1.0 ;

        /* Overall stopping condition for the algorithm */
        if(JT->time > 13.9 && JT->time < 16.1)
        {
                /* Terminate algorithm function when optimal solution is found */
                exec_terminate("ball_pre_master","End Condition reached.");
        }
        else
        {
               /* For bug checking purposes */
             //printf("I am in pre_master function.\n");
        }

        /* Assign the next best optimal time to the sim */
        B->out.time_to_fire_jet = JT->time ;

        return (0) ;
}
```

The “post” jobs are necessary for master/slave communication, which uses a TCP/IP connection. Obviously this link between the master and the slave simulations are important so that the master simulation can keep updating its variables until the optimal solution is reached.

```
/***************************** TRICK HEADER ********************************
PURPOSE:                (Read slave sim evaluationn)
****************************************************************************/

/* Necessary header files */
#include <stdio.h>
#include "ball/L1/include/ball_state.h"
#include "../include/jet_time.h"
#include "sim_services/include/executive.h"
#include "sim_services/include/exec_proto.h"

int ball_post_master( BSTATE* B,
                      JET_TIME* JT )
{
        BSTATE B_curr ;
        EXECUTIVE* E ;
        E = exec_get_exec() ;
        /* Read slave's results */
        tc_read( &E->monte.work.data_conn, (char*) &B_curr, sizeof(BSTATE) );

        /* Print to check increment */
        printf( "Time to fire is at:    %lf\n", JT->time );

        return (0);
}

/******************* TRICK HEADER ****************************
PURPOSE:        (Pass slave sim's evaluation of x to master)
***********************************************************/

#include "ball/L1/include/ball_state.h"
#include "sim_services/include/executive.h"
#include "sim_services/include/exec_proto.h"

int ball_post_slave ( BSTATE* B )
{
        EXECUTIVE* E ;
        E = exec_get_exec();

        /* Send F(x) - which is in BSTATE */
        tc_write(&E->monte.work.data_conn,(char*) B, sizeof(BSTATE));

        return(0);
}
```

The slave functions use Trick's Communication library (tc_write/tc_read) to pass information between the master and slave simulations.

4.3 The Simulation Files

4.3.1 S_define

The most obvious file we have to modify is the S_define since we have to include all of the functions that we added as well as a new simulation object for the optimization model that we created. We have to first include the jet functions that we created in Section 4.1 so that the simulation now takes the forces created by the jet. At the bottom of the ball sim_object, we add our two scheduled jet jobs.

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

...

 /*--- ADDED JET JOBS ---*/
 (0.1, scheduled) ball/L1:
        ball_force_jet(
                     Inout      BSTATE *S = &ball.state ) ;

 (0.1, scheduled) ball/L1:
        ball_jet_control(
                     Inout      BSTATE *S  = &ball.state ) ;
 /*-----------------------*/
} ball ; /*------------------------------------------------------------*/
```

However, for the simulation to actually take the forces created by the jet into account, we have to add a line to the collect statement at the bottom of the S_define file.

```
collect ball.state.work.external_force = { ball.force.out.force[0],
                                           ball.state.out.force_jet[0] } ;
```

And to finish up our modification with the S_define, we include another simulation object for the optimization model that we created.

```
sim_object /************ OPTIMIZATION OBJECT ***************/
{
        /* Data Structure Declarations */
        optim:  JET_TIME jet ;

        /* Optimization Initialization and Pre Jobs */
        (monte_master_init)     optim:
                optim_init( JET_TIME* JT = &optimizer.jet ) ;

        (monte_master_pre)      optim:
                ball_pre_master( BSTATE* S = &ball.state,
                                 JET_TIME* JT = &optimizer.jet ) ;

        /* Post Optimization Jobs */
        (monte_master_post)     optim:
                ball_post_master( BSTATE* S = &ball.state,
                                  JET_TIME* JT = &optimizer.jet ) ;

        (monte_slave_post)      optim:
                ball_post_slave( BSTATE* S = &ball.state ) ;

} optimizer ;
```

Now, the S_define has all the necessary modifications made to it, but since the Optimization framework that we're using from the Trick Tutorial is a tweaked form of Monte Carlo, we still have include the Monte Carlo input file and also add lines to the input file [2].

4.3.2 Input files

For the input file, we added three lines and changed the stopping time from 300 seconds to 20 seconds, just so we can see the changes that we made more easily in data plots. Below are the changes that we made:

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

stop = 20.0 ;
```

Since the this input file invokes the Monte Carlo input file *M_jet_firings_optim*, we also need to create this in the directory above the RUN_* directory:

```
NUM_RUNS: 300

VARS:

ball.state.out.time_to_fire_jet {s} CALC ;

DATA:
```

The variables in this input file just tells the simulation that those variables are going to be calculated by the simulation during each run, which is what one of our goals was for this project.

4.4 Running the Simulation

After all of the above elements are added, the ball simulation now has an optimization framework in place and it each run shows a change in the optimal time that the simulation fires the jet. The very last run is the run that satisfies the optimal ending condition, which is in this case trivial (when it reaches a certain time the simulation terminates).

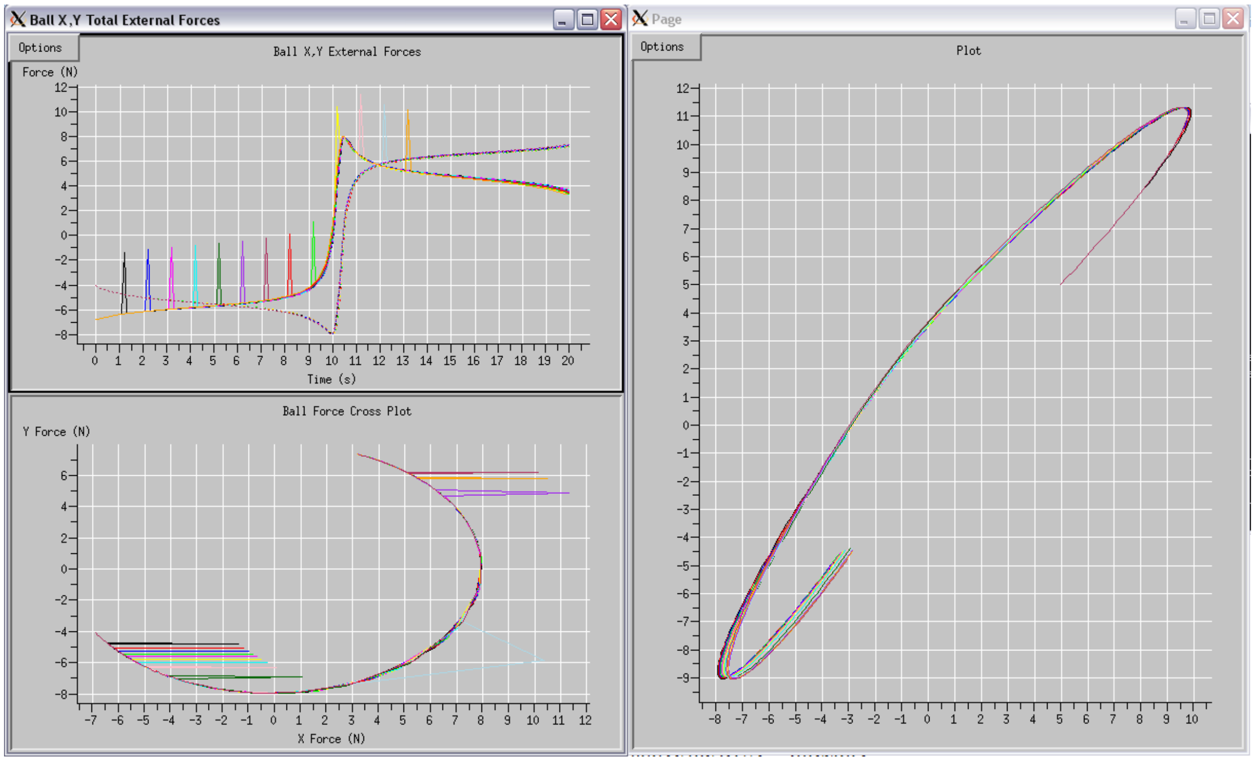

Figure 2 Force/Position Plot of Ball Simulation

Each color represents a different run and with each run, we can see that it the optimization framework increments the “optimal” time to fire the jet by one each run. From these graphs we can see that the optimization “algorithm” is indeed working.

5.0 C++ Implementation

Now that we have the C implementation of the L1 ball simulation with the optimization framework, the next hurdle would be to transfer this into C++. Since we have an existing L1 ball simulation in C++, a lot of the groundwork is there for us to use, but we still have to make worry about the new object oriented programming that comes along with C++.

5.1 The Jet in C++

5.1.1 The Jet Variables

Like the C implementation, we'll be start by adding the variables for the jet into the necessary header and data files. Again, adding the jet allows to easily optimize a variable and see the results on the graph.

In *Ball.dd* :

```
...

/* Added Jet data */
Ball.state.output.force_jet_plus {N} = 5.0 ;
Ball.state.output.time_to_fire_jet {s} = 0.0 ;
```

In *Ball_State.hh* :

```
...

class BallStateOutput {

  ...

  /* Jet variables */

  int jet_on ;                  /* -- 0|1 Jet firing? */
  int jet_count ;               /* -- How many jet firings? */
  double force_jet[3] ;         /* N force of jet */
  double force_jet_plus ;       /* N Configurable force of jet in +X direction */
  double time_to_fire_jet ;     /* s jet fire time */
};

...
```

These additions are almost exactly the same as the previous section; however, the following section does show that there are a few important differences that need to be taken into consideration when using C++ instead of C.

5.1.2 The Jet Function and Library Depedencies

We created one file, *BallJet.cpp*, that contains the two jet functions that were created in the previous implementation. These two jet functions, *ball_jet_control* and *ball_force_jet*, are extremely similar to the C functions except for some minor differences that arise due to the object oriented nature of C++. (The actual code can be found in the Appendix)

However, we have to make sure that the header file for the Ball class includes these two functions and has the proper library dependencies in order to use these two functions, which means that we have to modify the *Ball.hh* file:

```
/******************************** TRICK HEADER ******************************
PURPOSE:
    (Ball model EOM state parameter definition.)
...

LIBRARY DEPENDENCY:
    ((Ball.o)
     (BallStateDeriv.o)
     (BallStateInit.o)
     (BallStateInteg.o)
     (BallForceField.o)
     (BallJet.o))
...

****************************************************************************/

#ifndef _BALL_HH_
#define _BALL_HH_

...

class Ball {

  ...

   // Jet Functions (Fires and Controls the jets on the ball)
   int ball_force_jet();
   int ball_jet_control();
};

#endif /* _BALL_HH_ */
```

Under the Trick header in Library Dependency, it is important to include *BallJet.o* or else the code will compile with an undefined reference error. Also, the two functions that were created before need to also be included the Ball class.

5.2 Optimization in C++

There are two significantly different files that have to be created in a C++ version of optimization. For our project, we decided to create an Optimization class, just because it was easier for us to break it down and understand.

If you recall, in our previous C Implementation of optimization, our header file was *jet_time.h*, which consisted of only a single variable in its own data structure. However, since we decided to create an Optimization class, we have to do much more than a one variable data structure. The header file for the Optimization class, *Optimization.hh*, encapsulates all the optimization functions and the optimization variable, *optim_time*:

```
/************************ 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_time;   /* s optimal time to fire jet */


       // Optimization Initialization
       int optim_init();

       // The "pre" job function. This contains the optimization
       // algorithm and stopping conditions.
       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
```

Notice that like in *Ball.hh*, we still have to include the library dependencies for all of our C++ files that contain our optimization functions. These functions won’t be shown because the similarities to the C Implementation version are very close. (For the exact coding of these functions, please refer to the Appendix)

Another file, *Optimization.cpp*, which basically contains the constructor and the destructor, has to be created so that an optimization object is created:

```
/********************** TRICK HEADER *************************
PURPOSE:
       ( Optimization constructor/destructor )
LIBRARY DEPENDENCY:
       ((Optimization.o))
*************************************************************/

/* include files */
#include <iostream>
using namespace std;

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

// Default constructor
Optimization::Optimization()
{
       cout << "In Optim constructor." << endl;
}

// Destructor
Optimization::~Optimization()
{
       cout << "In Optim destructor." << endl;
}
```

5.3 Simulation Files with C++ models

5.3.1 S_define

The S_define goes through a lot of the same changes that was previously shown in the Section 4.0 of this report: The jets have to now be taken into account in the simulation, a new optimization object has to be created, and the forces from the jet need to be collected.

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

...

   /*------ JET JOBS ------*/

   (0.1, scheduled) Ball++/L1: ball.obj.ball_force_jet();

   (0.1, scheduled) Ball++/L1: ball.obj.ball_jet_control();

   /*-----------------------*/

} 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;


collect ball.obj.state.work.external_force = {ball.obj.force.output.force[0],
                                         ball.obj.state.output.force_jet[0]};

integrate (0.01) ball;
```

Obviously, there are only very minor modifications that were made to the S_define in order to incorporate the C++ models.

5.3.2 The input files

The *input* file is exactly the same as the one that was shown in Section 4.0 while the *M_jet_firings_optim* has very slight modifications to its VARS fieldv:

```
VARS:

ball.obj.state.output.time_to_fire_jet {s} CALC
```

5.4 Running the Simulation

The simulation should now be complete and should give exactly the same results as the C Implementation of this simulation, which was the goal of this section. However, if you try to execute the simulation and you get a “glibc detected” with a segmentation fault, then you might have to type the following command in the terminal in order to fix this error:

```
export MALLOC_CHECK_=0
```

Why this occurs is still unknown, but it is definitely something that should be looked into in the future. The resulting data plot should look exactly the same as before:

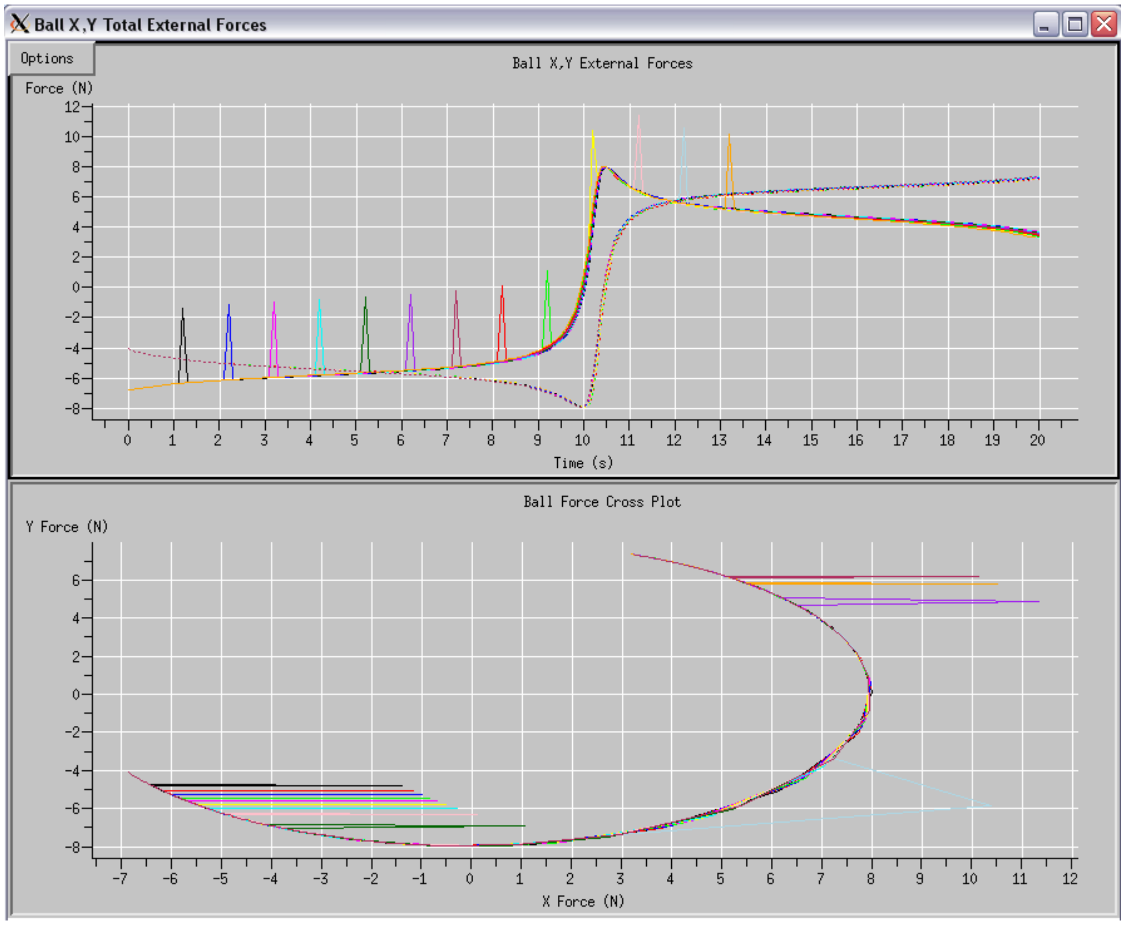

Figure 3 Force Plot using C++ model files

5.5 Adding Optimization Variables

Now that we have the framework set up, we will show how to add new variables for optimization. For this section, we will be modifying the initial elevation of the ball. As always, first thing is first and we created a local variable that represents the variable that we want to be optimized.

In Optimization.hh, we add *optim_elev*:

```
// Optimization variables
double optim_time;      /* s optimal time to fire jet */
double optim_elev;      /* r optimal trajectory angle */
```

And in Optimization.dd, we include its initial condition (which may actually be anything since it will actually get its value from the actual variable in the L1 ball simulation):

```
Optimization.optim_elev {d} = 0.0
```

The next step would be to initialize that variable in the *OptimInit.cpp* file, but since there really is no reason to initialize the elevation variable, we just skipped this step.

For the algorithm stage in the *OptimPreJobs.cpp* file, we decided that we would just add 10 degrees to the elevation through each run (similar to adding 1 s to the firing time in the previous sections), just so that we could see the results on a plot easily. We keep everything pretty much the same as before (the ending condition is actually still dependent on time, but this can also be changed easily).

The first modification that needs to be done to the function is to assign the local variable to the actual variable from the other simulation and then increment the local variable by 10 degrees. Then, we just assign the new value of the local variable to the original variable.

The absolute stopping condition was also modified to check the elevation angle since this was the new variable of concern. The new stopping condition results in four iterations of increasing trajectory angles before the program stops.

```
int Optimization::ball_pre_master(Ball *B)
{
        // Increment "optim" time by 1 second. Note that this is
        // the "algorithm"
        this->optim_time += 1.0 ;

        // Change initial elevation and increase by 10 degrees
        this->optim_elev = B->state.input.elevation ;
        this->optim_elev += (10*3.14/180) ;

        // Absolute Stopping Condition
        if(this->optim_elev > (30*(3.14/180))
        {
                // Terminates this function once the end condition is reached
                exec_terminate("ball_pre_master","End Condition reached.");
        }
        else
        {
                cout << "I am in pre_master function.\n" << endl;
        }

        // Updates the "optimal" time and elevation
        B->state.output.time_to_fire_jet = this->optim_time ;
        B->state.input.elevation = this->optim_elev ;

        return (0) ;
}
```

Notice that the modifications come easily because the elevation variable is part of the *Ball* class (it is also equally important to realize that in the post jobs we are sending the *Ball* class over the connection), and if we wanted to modify a variable from another class, we would have to make more extensive modifications.

Ball.dd was also modified to provide initial conditions that result in a less chaotic trajectory:

In *Ball.dd* :

```
...

Ball.force.input.origin[0] {M} = 0.0, 0.0 ;
Ball.force.input.force {N} = 8.0 ;

Ball.state.input.mass {kg} = 10.0 ;
Ball.state.input.speed {M/s} = 2.00 ;
Ball.state.input.elevation {d} = -10.0 ;
Ball.state.input.position[0] {M} = 0.0 , 5.0 ;


/* Added Jet data */
Ball.state.output.force_jet_plus {N} = 0.0 ;
Ball.state.output.time_to_fire_jet {s} = 0.0 ;
```

The position graph, due to the changing elevations, is now supposed to be like the figure below, showing a trajectory with an initial angle that increases by 10 degrees each iteration (initial trajectory in black), with each iteration plotted over a period of 3 seconds:

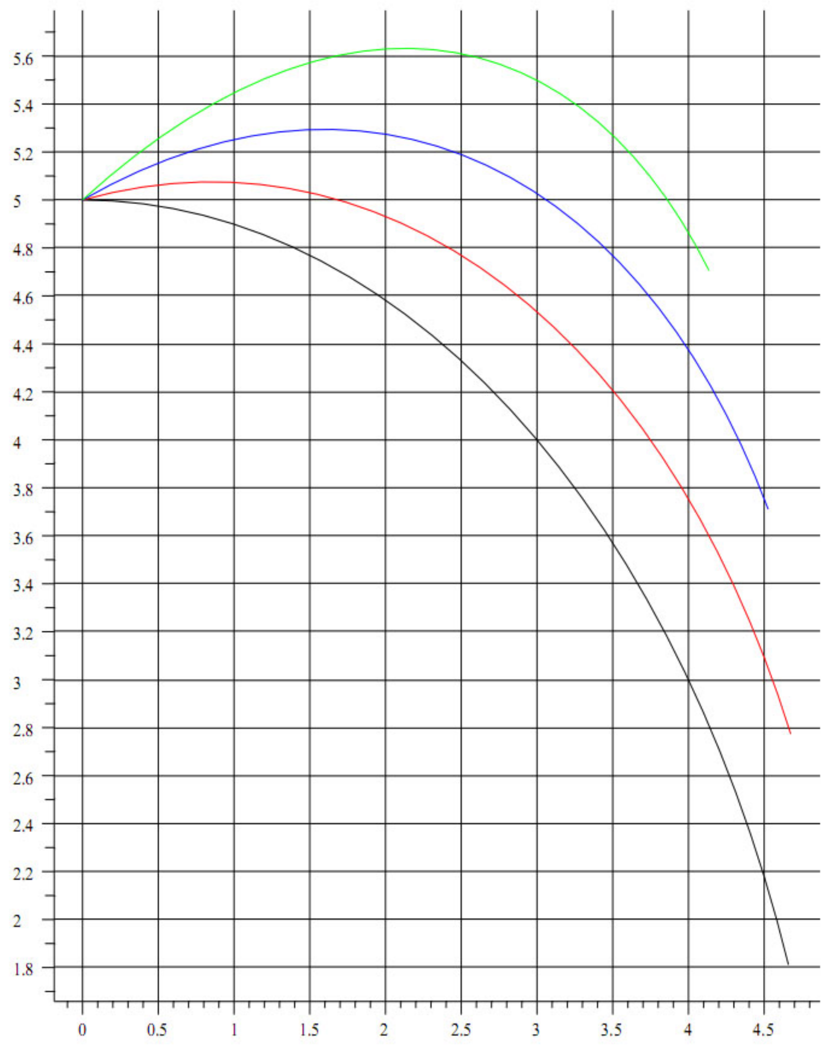

Figure 4 Position plot with changing elevations

6.0 Future Work

With the implementation of Trick’s optimization framework into a simple C++ simulation completed, the next step will be to apply what was learned to an orbital targeting program using NASA’s JEOD system and an independently developed targeting algorithm.   This will entail learning how to use JEOD, and then using what was learned from both the work done with Trick’s optimization system and experimentation with JEOD to contribute to the more complex orbital targeting program already in development at the University of Texas.   The primary obstacle will likely be encountered in learning how to use JEOD, due to its apparent complexity and lack of thorough documentation.   Integrating the optimization code is not expected to be a problem due to what was learned during this project.

7.0 References

[1] Vetter, Keith and Grace Hua. *The Trick User's Guide*, 2005.

[2] Vetter, Keith and Grace Hua. *Trick Simulation Environment User Training Materials*, 2005.

8.0 Appendix

Trick includes sample simulations in its folders. This appendix only contains files that have been modified for this project. The sample model files included with trick can be found at $TRICK_HOME/trick_models and sample simulation files can be found in $TRICK_HOME/trick_sims. The optim and Optim++ model directories were created for this project, so all source code from those folders will be included below. For more information on the master/slave framework and Optimization, please refer to Section 13.0 of the Trick Tutorial [2].

Note: Revision log notes have been removed from the source code for brevity.

8.1 Modified C model code

8.1.1 trick_models/ball/L1/include

ball_state.d

```
/********************************* TRICK HEADER *******************************
PURPOSE:
    (Ball model EOM state parameter 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:
    ((2-D Space)
     (Translational EOM only))
PROGRAMMERS:
   (((Your Name) (Company Name) (Date) (Trick tutorial)))
*******************************************************************************/

BSTATE.in.mass {kg} = 10.0 ;
BSTATE.in.speed {M/s} = 3.5 ;
BSTATE.in.elevation {d} = 45.0 ;
BSTATE.in.position[0] {M} = 5.0 , 5.0 ;

/* Initial data needed for jet */
BSTATE.out.force_jet_plus  {N} = 5.0 ;
BSTATE.out.time_to_fire_jet {s} = 0.0 ;
```

ball_state.h

```
/********************************* 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:
    ((2 dimensional space)
     (Translational EOM only))
PROGRAMMERS:
    (((Robert W. Bailey) (Sweet Systems Inc) (March 1997) (Tutorial Lesson 1)))
*******************************************************************************/

#ifndef _BALL_STATE_H_
#define _BALL_STATE_H_

typedef struct { /* BSTATE_IN ------------------------------------------------*/

  /*=== Initial Ball States ===*/
  double mass ;            /* kg  Total mass */
  double position[2] ;     /* M   X(horizontal),Y(vertical) position */
  double speed ;           /* M/s Linear speed */
  double elevation ;       /* r   Trajectory angle with respect
                                     to the horizontal */
} BSTATE_IN ; /*--------------------------------------------------------------*/

typedef struct { /* BSTATE_OUT -----------------------------------------------*/

  double position[2] ;       /* M    X(horizontal), Y(vertical) position */
  double velocity[2] ;       /* M/s  X,Y velocity */
  double acceleration[2] ;   /* M/s2 X,Y acceleration */
  double external_force[2] ; /* N    Total external force on ball */


  /* Variables needed for Jet addition */
  int jet_on ;                 /* -- 0|1 Jet firing? */
  int jet_count ;              /* -- How many jet firings? */
  double force_jet[3] ;        /* N force of jet */
  double force_jet_plus ;      /* N Configurable force of jet in +X direction */
  double time_to_fire_jet ;    /* s jet fire time */
} BSTATE_OUT ; /*-------------------------------------------------------------*/

typedef struct { /* BSTATE_WORK ----------------------------------------------*/

  void ** external_force ;   /* ** N    External forces, from 'collect' */

} BSTATE_WORK ; /*------------------------------------------------------------*/

typedef struct { /* BSTATE ---------------------------------------------------*/

  BSTATE_IN   in ;           /*    --   User inputs */
  BSTATE_OUT  out ;          /*    --   User outputs */
  BSTATE_WORK work ;         /*    --   EOM workspace */

} BSTATE ; /*-----------------------------------------------------------------*/

#endif
```

8.1.2 trick_models/ball/L1/src

ball_state_init.c

```
/******************************** TRICK HEADER ******************************
PURPOSE:
    (ball_state_init performs the following:
        - passes the input position vector to the output vector
        - xforms the input ball speed and trajectory elevation into a
          velocity vector.)
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)
     (Positive X is horizontal to the right)
     (Positive Y is vertical and up))
CLASS:
    (initialization)
LIBRARY DEPENDENCY:
    ((ball_state_init.o))
PROGRAMMERS:
    (((Your Name) (Company Name) (Date) (Trick tutorial)))
*****************************************************************************/

/* SYSTEM INCLUDE FILES */
#include <math.h>

/* GLOBAL DATA STRUCTURE DECLARATIONS */
#include "../include/ball_state.h"

/* ENTRY POINT */
int ball_state_init(
                    /* RETURN: -- Always return zero */
                    BSTATE *S ) /* INOUT:  -- Ball EOM state parameters */
{

     /* GET SHORHAND NOTATION FOR DATA STRUCTURES */
     BSTATE_IN  *BI = &(S->in) ;
     BSTATE_OUT *BO = &(S->out) ;

     /* TRANSFER INPUT POSITION STATES TO OUTPUT POSITION STATES */
     BO->position[0] = BI->position[0] ;     /* X state */
     BO->position[1] = BI->position[1] ;     /* Y state */

     /* TRANSFER INPUT SPEED AND ELEVATION INTO THE VELOCITY VECTOR */
     BO->velocity[0] = BI->speed * cos( BI->elevation ) ;       /* X velocity */
     BO->velocity[1] = BI->speed * sin( BI->elevation ) ;       /* Y velocity */

     /* FORCE OF JET */
     S->out.force_jet_plus = 5.0 ;
     /* RETURN */
     return( 0 ) ;
}
```

ball_jet_control.c

```
/************************ TRICK HEADER *******************************
PURPOSE:                ( Controls when jets are fired )
*********************************************************************/

#include "../include/ball_state.h"
#include "sim_services/include/exec_proto.h"
#include "trick_utils/math/include/trick_math.h"

/* Define what equality is */
#define BALL_EQUALS(X,Y) ( fabs(X-Y) < 1.0e-9 ) ? 1 : 0

int ball_jet_control( BSTATE* B )
{
        /* Declare varaible for simulation time */
        double sim_time ;
        sim_time = exec_get_sim_time() ;

        /* If the simulation time equals the time that the jet fires, then
           turn the jet on */
        if(BALL_EQUALS(sim_time, roundoff(.1, B->out.time_to_fire_jet)))
        {
                B->out.jet_on = 1 ;
        }
        return(0) ;
}
```

ball_force_jet.c

```
/******************************** TRICK HEADER ***********************
PURPOSE:                ( Jet fire force )
*******************************************************************/

#include "../include/ball_state.h"

int ball_force_jet( BSTATE *B )
{
        if ( B->out.jet_on )
        {
                /* If jet is on, then assign the jet a certain force
                   and then turn the jet off. */
                B->out.force_jet[0] = B->out.force_jet_plus ;
                B->out.jet_on = 0 ;
        }
        else
                B->out.force_jet[0] = 0.0 ;

        return ( 0 );
}
```

8.1.3 trick_models/optim/include

jet_time.h

```
/****************************** TRICK HEADER **************************
PURPOSE:                (Optimal time to fire the jet)
*********************************************************************/

#ifndef _JET_TIME_H_
#define _JET_TIME_H_

typedef struct {

        double time;    /* s JET_TIME */

} JET_TIME ;

#endif
```

8.1.4 trick_models/optim/src

optim_init.c

```
/******************************* TRICK HEADER ************************************
PURPOSE:                (Pre master algorithm to modify ball position)
*******************************************************************************/

#include "ball/L1/include/ball_state.h"
#include "../include/jet_time.h"

int optim_init( JET_TIME* JT )
{
        /* Initializes variables needed for the optimization algorithm */
        JT->time = 0.0 ;

        return (0) ;
}
```

ball_pre_master.c

```
/******************************* TRICK HEADER ************************************
PURPOSE:                (Pre master algorithm to modify ball position)
*******************************************************************************/

#include "ball/L1/include/ball_state.h"
#include "../include/jet_time.h"

int ball_pre_master(BSTATE* B,
                    JET_TIME* JT)
{
        /* "Algorithm" that increments the optimal time by 1 */
        JT->time += 1.0 ;

        /* Overall stopping condition for the algorithm */
        if(JT->time > 13.9 && JT->time < 16.1)
        {
                /* Terminate algorithm function when optimal solution is found */
                exec_terminate("ball_pre_master","End Condition reached.");
        }
        else
        {
               /* For bug checking purposes */
             //printf("I am in pre_master function.\n");
        }

        /* Assign the next best optimal time to the sim */
        B->out.time_to_fire_jet = JT->time ;

        return (0) ;
}
```

ball_post_master.c

```
/****************************** TRICK HEADER *********************************
PURPOSE:                (Read slave sim evaluationn)
*****************************************************************************/

/* Necessary header files */
#include <stdio.h>
#include "ball/L1/include/ball_state.h"
#include "../include/jet_time.h"
#include "sim_services/include/executive.h"
#include "sim_services/include/exec_proto.h"

int ball_post_master( BSTATE* B,
                      JET_TIME* JT )
{
        BSTATE B_curr ;
        EXECUTIVE* E ;
        E = exec_get_exec() ;

        /* Read slave's results */
        tc_read( &E->monte.work.data_conn, (char*) &B_curr, sizeof(BSTATE) );

        /* Print to check increment */
        printf( "Time to fire is at:    %lf\n", JT->time );

        return (0);
}
```

ball_post_slave.c

```
/******************** TRICK HEADER ****************************
PURPOSE:        (Pass slave sim's evaluation of x to master)
```

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

#include "ball/L1/include/ball_state.h"
#include "sim_services/include/executive.h"
#include "sim_services/include/exec_proto.h"

int ball_post_slave ( BSTATE* B )
{
        EXECUTIVE* E ;
        E = exec_get_exec();

        /* Send F(x) - which is in BSTATE */
        tc_write(&E->monte.work.data_conn,(char*) B, sizeof(BSTATE));

        return(0);
}
```

8.2 Simulation files needed for the ball and optim models (in C)

8.2.1 trick_sims/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:  BSTATE      state  (ball/L1/include/ball_state.d) ;
             ball/L1:  BFORCE      force  (ball/L1/include/ball_force.d) ;
 sim_services/include:  INTEGRATOR  integ  (ball/L1/include/ball_integ.d) ;

 /*----- INITIALIZATION JOBS -----*/
 (initialization) ball/L1:ball_state_init(
                    Inout     BSTATE *S   = &ball.state ) ;

 /*----- EOM DERIVATIVE AND STATE INTEGRATION JOBS -----*/
 (derivative) ball/L1:ball_force_field(
                    Inout     BFORCE *F   = &ball.force ,
                    In        double *pos = ball.state.out.position ) ;
 (derivative) ball/L1:ball_state_deriv(
                    Inout     BSTATE *S   = &ball.state ) ;
 (integration) ball/L1:ball_state_integ(
                    Inout INTEGRATOR *I   = &ball.integ ,
                    Inout     BSTATE *S   = &ball.state ) ;

 /*--- ADDED JET JOBS ---*/
 (0.1, scheduled) ball/L1:
        ball_force_jet(
                     Inout      BSTATE *S = &ball.state ) ;

 (0.1, scheduled) ball/L1:
        ball_jet_control(
                     Inout      BSTATE *S  = &ball.state ) ;
 /*----------------------*/
} ball ; /*----------------------------------------------------------*/


sim_object /************ OPTIMIZATION OBJECT ***************/
{
        /* Data Structure Declarations */
        optim:  JET_TIME jet ;

        /* Optimization Initialization and Pre Jobs */
        (monte_master_init)     optim:
                optim_init( JET_TIME* JT = &optimizer.jet ) ;

        (monte_master_pre)      optim:
                ball_pre_master( BSTATE* S = &ball.state,
                                 JET_TIME* JT = &optimizer.jet ) ;

        /* Post Optimization Jobs */
        (monte_master_post)     optim:
                ball_post_master( BSTATE* S = &ball.state,
                                  JET_TIME* JT = &optimizer.jet ) ;

        (monte_slave_post)      optim:
                ball_post_slave( BSTATE* S = &ball.state ) ;

} optimizer ;



collect ball.state.work.external_force = { ball.force.out.force[0],
                                           ball.state.out.force_jet[0] } ;

integrate (0.01) ball ;
```

M_jet_firings_optim

```
NUM_RUNS: 300

VARS:

ball.state.out.time_to_fire_jet {s} CALC ;

DATA:
```

8.2.2 trick_sims/SIM_ball_L1/RUN_test

input

```
/* Created By: bailey
         Date: Mon Apr 14 1997 06:16:00 PM
*/

#include "S_default.dat"
```

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

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

stop = 20.0 ;
```

8.3 Modified C++ model code

8.3.1 trick_models/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)))
*******************************************************************************/

Ball.force.input.origin[0] {M} = 0.0, 2.0 ;
Ball.force.input.force {N} = 8.0 ;

Ball.state.input.mass {kg} = 10.0 ;
Ball.state.input.speed {M/s} = 3.5 ;
Ball.state.input.elevation {d} = 45.0 ;
Ball.state.input.position[0] {M} = 5.0 , 5.0 ;

/* Added Jet data */
Ball.state.output.force_jet_plus {N} = 5.0 ;
Ball.state.output.time_to_fire_jet {s} = 0.0 ;
```

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:
    ((2 dimensional space)
     (Translational EOM only))
LIBRARY DEPENDENCY:
    ((Ball.o)
     (BallStateDeriv.o)
     (BallStateInit.o)
     (BallStateInteg.o)
     (BallForceField.o)
     (BallJet.o))
PROGRAMMERS:
    (((Robert W. Bailey) (Sweet Systems Inc) (March 1997) (Tutorial Lesson 1))
     ((Edwin Z. Crues)(Titan Systems Corp.)(Jan 2002)(Crude C++ translation)))
*****************************************************************************/

#ifndef _BALL_HH_
#define _BALL_HH_

// System include files.

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

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


class Ball {

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

   // Initialization functions.
   int state_init();

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

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

   // Jet Functions (Fires and Controls the jets on the ball)
   //int ball_force_jet();
   //int ball_jet_control();

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

   // Jet Functions (Fires and Controls the jets on the ball)
   int ball_force_jet();
   int ball_jet_control();
};

#endif /* _BALL_HH_ */
```

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:
    ((2 dimensional space)
     (Constant force)
     (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)))
*****************************************************************************/

#ifndef _BALL_STATE_HH_
#define _BALL_STATE_HH_

class BallStateInput {
 public:
  double mass;        /* *i kg  Total mass.                           */
  double position[2]; /* *i M   X(horizontal), Y(vertical) position. */
  double speed;       /* *i M/s Linear speed.                         */
  double elevation;   /* *i r   Trajectory angle with respect
                                to the horizontal.                    */
};

class BallStateOutput {
 public:
  double position[2];       /* *o M    X(horizontal), Y(vertical) position. */
  double velocity[2];       /* *o M/s  X, Y velocity.                        */
  double acceleration[2];   /* *o M/s2 X, Y acceleration.                    */
  double external_force[2]; /* *o N    Total external force on ball.         */

  /* Jet variables */

  int jet_on ;                  /* -- 0|1 Jet firing? */
  int jet_count ;               /* -- How many jet firings? */
  double force_jet[3] ;         /* N force of jet */
  double force_jet_plus ;       /* N Configurable force of jet in +X direction */
  double time_to_fire_jet ;     /* s jet fire time */
};

class BallStateWork {
 public:
   void ** external_force; /* ** N External forces, from 'collect' */
};

class BallState {

  public:
   /* Member data. */
   BallStateInput  input;  /* -- User inputs    */
   BallStateOutput output; /* -- User outputs.  */
   BallStateWork   work;   /* -- EOM workspace. */

};

#endif /* _BALL_STATE_HH_ */
```

8.3.2 trick_models/Ball++/L1/src

BallJet.cpp

```
/************************ TRICK HEADER *******************************
PURPOSE:
        ( Jet firing force and Jet Control )
LIBRARY DEPENDENCY:
       ((BallJet.o))
*********************************************************************/

// include class headers
#include "../include/Ball.hh"

// include math and exec libraries
#include "sim_services/include/exec_proto.h"
#include "trick_utils/math/include/trick_math.h"

// define equality (doubles are rarely perfectly equal)
#define BALL_EQUALS(X,Y) ( fabs(X-Y) < 1.0e-9 ) ? 1 : 0

int Ball::ball_force_jet()
{
       /* Shorthand notation for data structures */
       BallStateOutput * state_out = &(this->state.output);

       if ( state_out->jet_on )
        {
              /* If the Jet is On, then assign a force to the
                 jet and then reassign to zero for next run */

              state_out->force_jet[0] = state_out->force_jet_plus ;

              /* Print statements just to verify */
                printf("%lf\n",state_out->force_jet_plus);
                printf("%lf\n",state_out->force_jet[0]);

              /* Reset to zero */
                state_out->jet_on = 0 ;
        }
        else
                state_out->force_jet[0] = 0.0 ;

        return (0) ;
}


int Ball::ball_jet_control()
{
       /* Assign the Simulation time variable */
        double sim_time ;
        sim_time = exec_get_sim_time() ;

       /* Shorthand notation for data structures */
       BallStateOutput * state_out = &(this->state.output);

       /* If the sim time is equal to the time that the jet is supposed to
          fire then that means the jet is on */
        if(BALL_EQUALS(sim_time, roundoff(.1, state_out->time_to_fire_jet)))
        {
                state_out->jet_on = 1 ;
        }
        return(0) ;
}
```

8.3.3 trick_models/Ball++/Optim++/include

Optimization.dd

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

Optimization.optim_time {s} = 0.0 ;
```

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_time;   /* s optimal time to fire jet */


       // Optimization Initialization
       int optim_init();

       // The "pre" job function. This contains the optimization
       // algorithm and stopping conditions.
       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
```

8.3.4 trick_models/Ball++/Optim++/src

Optimization.cpp

```
/********************** TRICK HEADER *************************
PURPOSE:
       ( Optimization constructor/destructor )
LIBRARY DEPENDENCY:
       ((Optimization.o))
*************************************************************/

/* include files */
#include <iostream>
using namespace std;

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

// Default constructor
Optimization::Optimization()
{
       cout << "In Optim constructor." << endl;
}

// Destructor
Optimization::~Optimization()
{
       cout << "In Optim destructor." << endl;
}
```

OptimInit.cpp

```
/********************************* TRICK HEADER ***********************************
PURPOSE:
       (Pre master algorithm to modify ball position)
LIBRARY DEPENDENCY:
       ((OptimInit.o))
**********************************************************************************/

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

int Optimization::optim_init()
{
       /* Set Initial Conditions for Optimization Algorithm */
        this->optim_time = 0.0 ;

       /* NOTE: If only variables need to be set, it might be
              better to just use a .dd file instead. If
              calculations are needed for the algorithm, then
              this initial function call is needed */

        return (0) ;
}
```

OptimPreJobs.cpp

```
/*************************** TRICK HEADER ********************************
PURPOSE:      ( Contains the algorithm and end condition )
*************************************************************************/

#include <iostream>
```

```
using namespace std;

/* Trick Headers */
#include "sim_services/include/exec_proto.h"

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

int Optimization::ball_pre_master(Ball *B)
{
       // Increment "optim" time by 1 second. Note that this is
       // the "algorithm"
        this->optim_time += 1.0 ;

       // Absolute Stopping Condition
        if(this->optim_time > 13.9 && this->optim_time < 16.1)
        {
              // Terminates this function once the end condition is reached
                exec_terminate("ball_pre_master","End Condition reached.");
        }
        else
        {
                cout << "I am in pre_master function.\n" << endl;
        }

       // Updates the "optimal" time
        B->state.output.time_to_fire_jet = this->optim_time ;

        return (0) ;
}
```

OptimPostJobs.cpp

```
/****************************** TRICK HEADER *********************************
PURPOSE:         ( Optimization Post Jobs, contains TCP/IP Comm )
*****************************************************************************/

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

#include "sim_services/include/executive.h"
#include "sim_services/include/exec_proto.h"

int Optimization::ball_post_master()
{
        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) );

        printf( "Time to fire is at:    %lf\n", this->optim_time );

}


int Optimization::ball_post_slave ( Ball* B )
{
        EXECUTIVE* E ;
        E = exec_get_exec();

        /* Send F(x) - which is in BSTATE */
        tc_write(&E->monte.work.data_conn,(char*) B, sizeof(Ball));

        return(0);
}
```

8.4 Simulation files needed for the Ball++ and Optim++ models (in C++)

8.4.1 trick_sims/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.force_field();

   (derivative) Ball++/L1: ball.obj.state_deriv();

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

   /*------ JET JOBS ------*/

   (0.1, scheduled) Ball++/L1: ball.obj.ball_force_jet();

   (0.1, scheduled) Ball++/L1: ball.obj.ball_jet_control();

   /*----------------------*/

} 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;

collect ball.obj.state.work.external_force = {ball.obj.force.output.force[0],
                                        ball.obj.state.output.force_jet[0]};

integrate (0.01) ball;
```

M_jet_firings_optim

```
NUM_RUNS: 300

VARS:

ball.obj.state.output.time_to_fire_jet {s} CALC ;

DATA:
```

8.4.2 trick_sims/SIM_Ball++_L1/RUN_test

input

```
/* Created By: bailey
         Date: Mon Apr 14 1997 06:16:00 PM
*/

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

stop = 20.0 ;

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