**TRANSMITAL MEMO**

| | |
|---|---|
| TM NO. | |
| DATE | |
| CONTRACT NO. | |
| TASK ORDER NO. | |

**SUBJECT:** **Trick Documentation**

**ENCLOSURE:** **Update to User Training Materials for the Trick 2013.0 Release**

**TO**
**NASA/LYNDON B JOHNSON SPACE CENTER**
**2101 NASA Parkway**
**HOUSTON, TEXAS 77058**
**ATTN: Alex Lin (ER7)**

**REMARKS:**

The responsibility for this document lies with the Simulation and Graphics Development Branch (ER7) of the Automation, Robotics, and Simulation Division (AR&SD) of the NASA JSC Engineering Directorate. This document applies to all Trick simulation developers and simulation users throughout NASA and its contractors.

**Modified by:**

Donna Panter
Senior Software Engineer

# Trick Simulation Environment User Training Materials Trick 2013.0 Release

Prepared by

Keith Vetter

Modified by

Donna Panter
L-3 Communications Corporation
1002 Gemini, Suite 200
Houston, Texas 77058

**NATIONAL AERONAUTICS AND SPACE ADMINISTRATION**

**JOHNSON SPACE CENTER**

**SOFTWARE, ROBOTICS and SIMULATION DIVISION**

This Page Intentionally Blank

# User Training Guide

# List of Figures

*Figure* *Page*

# List of Tables

*Table* *Page*

# 1.0 Introduction

Trick is a software package used to build and run simulations. Trick provides utilities and a simulation executive that work together to transform developer's model code into either a real-time simulation or a non-real-time simulation.

The Trick name is handed down from motocross. "Trick" isn't an acronym. It is slang for "really cool"... or something of that nature. :-)

This tutorial covers the basic concepts for developing and operating Trick simulations. It assumes absolutely no previous knowledge of Trick. The tutorial should be self-contained with no need for outside references other than maybe a C book, and/or a UNIX book. By the end of the tutorial, you should know how to build a Trick simulation of your own from the ground up.

The approach of the tutorial is hands-on. The intent is not to teach about simulation math models, nor to teach about C programming, nor UNIX, but rather to teach about the Trick architecture and its capabilities.

The example simulation used through this tutorial is a simulation of a flying cannonball (or Holy Hand Grenade). Up until 2004, the simulation had been of an orbiting ball in space. After many discussions with tutorial users it became apparent that the tutorial was leaving many lost. The intent of this revision is to make things simpler and clearer. We hope that some of Trick's "magic" will be dispelled.

We are very interested in any comments that you have about this tutorial. Please make use of the Trick bulletin board to post any problems, questions or suggestions. Reporting your problems can help the next person who follows to avoid any pitfalls. You can access the bulletin board from the Trick website: http://trick.jsc.nasa.gov.

Many thanks go to those who sent comments on the orbiting ball tutorial. Many thanks to those in the future who will send comments as well!

# 2.0 Becoming a Trick User

Assuming you have the Trick package, you must obtain a "Trick Environment" before creating Trick simulations. The environment is based on bash, csh, tcsh or an equivalent. Most Trick users use tcsh, so the tutorial will assume tcsh/csh. This section goes through the steps in setting up your environment.

This tutorial was designed for Trick versions 13.1.0 and later.

## 2.1 The New User Installation Script - install_user

To set up the UNIX environment necessary for building Trick simulations, you must execute a script called `install_user`. To find the `install_user` script it is necessary to know where Trick was installed. If Trick was installed with an RPM, it will be found in /usr/local/trick/<version>. If Trick isn't located in /usr/local/trick/<version>, it was probably built from source. If this is the case, you will have to fish for its location or find the person who installed it.

When you run the install user script you may safely accept the defaults for the tutorial purposes.

1. Start the user installation script/GUI and answer the questions:

   **UNIX Prompt>** `<Path To Trick Install Dir>/bin/install_user`

2. If you are using the csh or tcsh environment, add the following line to .cshrc in your home directory:
   ```
   source ~/.Trick_cshrc_<version>
   ```
3. If you are using bash, add the following line to .profile in your home directory:
   ```
   . ~/.Trick_profile_<version>
   ```
4. Logout of your workstation.
5. Log back in. You should now be an official, installed Trick user.
6. To verify that you have a Trick environment, enter the command as shown below. Once the "gte" command is issued you should see a spray of variables and settings (gte stands for "Get Trick Environment"). If this command fails, you have not obtained a Trick environment, and you may not proceed.

   **UNIX Prompt>** `gte`

# 3.0 Cannonball Simulation - A First Try

Now that you are a Trick user, you may begin creating simulations.

The cannonball simulation will be made in stages with differing approaches. This is the first approach, and definitely not the best. This first attempt will introduce you to general Trick concepts. We will improve the simulation with more advanced techniques as we go.

## 3.1 Cannonball Problem Stated

**Figure 1 Cannonball**

-g
init angle
init speed
impact point

A cannonball is fired from a cannon with an initial speed and initial angle. Simulate the trajectory of the cannonball. Assume no air friction. The only force to act on the ball is gravity.

## 3.2 Cannonball Problem Solved Without Trick

This is simple enough. Why not code this without a "simulation package"? Here is some code that will "simulate" the cannonball.

**Figure 2 Trickless Cannonball Code**

```
/* Cannonball without Trick */

#include <stdio.h>
#include <math.h>

int main() {

       /* Declare variables used in simulation */
       double pos[2]; double pos_orig[2] ;
       double vel[2]; double vel_orig[2] ;
       double acc[2];
       double init_angle ;
       double init_speed ;
       double time ;

       /* Initialize data */
       pos[0] = 0.0 ; pos[1] = 0.0 ;
       vel[0] = 0.0 ; vel[1] = 0.0 ;
       acc[0] = 0.0 ; acc[1] = -9.81 ;
       time = 0.0 ;
       init_angle = M_PI/6.0 ;
       init_speed = 50.0 ;

       /* Do initial calculations */
       pos_orig[0] = pos[0] ;
       pos_orig[1] = pos[1] ;
       vel_orig[0] = cos(init_angle)*init_speed ;
       vel_orig[1] = sin(init_angle)*init_speed ;

       /* Run simulation */
       while (  pos[1] >= 0.0 ) {

              vel[0] = vel_orig[0] + acc[0]*time ;
              vel[1] = vel_orig[1] + acc[1]*time ;

              pos[0] = pos_orig[0] + vel_orig[0]*time + (0.5)*acc[0]*time*time ;
              pos[1] = pos_orig[1] + vel_orig[1]*time + (0.5)*acc[1]*time*time ;

              time += 0.01 ;
       }

       /* Shutdown simulation */
       printf("Impact time=%lf position=%lf\n", time, pos[0]);

       return 0 ;
}
```

Voila! A cannonball simulation. Who needs Trick? Since our livelihood rests on the development of the Trick simulation package, we had better defend the need!

Actually, this simple main() program gives a glimpse of the overall architecture of Trick. The program could be divided into five distinct parts:

1. Declarations.
2. Data initialization.
3. Initialization routines.
4. Simulation executive.
5. Cleanup/Shutdown.

## 3.3 Where The Trickless Simulation Breaks Down

There are some problems in the hand-made simulation from section 3.2.

1. It doesn't give the correct answer! For one, the impact doesn't occur on a 0.01 second boundary.
2. This hand-made simulation architecture will not scale well. The bigger it gets the more brittle it will become. As it grows in size and complexity, it will be harder and harder to maintain. It will be more and more difficult to add new features, such as drag, jets or environment without major reorganization.
3. The hand-made cannonball simulation is fairly unique in that position is a function of time. How many real-world simulations have an analytical solution? Maybe two: perfect spring and perfect cannonball trajectory. In most simulations, we will need a way to integrate from one state to another.
4. This simulation does not run "real-time". It runs as quick as it can in its while() loop. There is no notion of synching with a clock.
5. This simulation leaves us no trail for analysis. There is no data recorded. No pretty plots.
6. In order to modify the angle of the cannon, we are forced to put a new value in the program and recompile! Wouldn't it be nice to be able to read the angle from an input file?
7. There is no notion of units in the simulation.

There may be more problems with the Trickless version. Seven would be a good stopping point.

After the discussion on the shortcomings of our hand-made simulation, we might add a few things to the list of things we want out of a simulation package:

1. Data recording.
2. Notion of real-time (with an option of running as fast as it can!).
3. Scalability.
4. Modularity.
5. Plotting package.
6. Input file processor so we can change our parameters without recompiling!
7. An integration scheme for propagating states.

## 3.4 Cannonball Simulated Using Trick

This tutorial is all about writing simulations with Trick. Why don't we try that?!? At this point you should be a Trick user. If not, see section 2.1.

Going back to our Trickless version we saw there were five parts: data declarations, data initialization, initialization routines, simulation executive and shutdown.

Here goes! We shall begin writing a cannonball simulation using Trick.

### 3.4.1 Creating A Directory System To Hold Simulation Source

We will begin by creating a directory system to hold our simulation source code

```
UNIX Prompt>  cd $HOME
UNIX Prompt>  mkdir -p trick_sims/SIM_cannon_analytic
UNIX Prompt>  mkdir -p trick_models/cannon/gravity/src
UNIX Prompt>  mkdir -p trick_models/cannon/gravity/include
```

It is standard practice, although not mandatory, to place sims in a "sims" directory, and model code in a "model" directory.

### 3.4.2 Data Declarations

At the top of our Trickless cannonball simulation we declared a few variables such as position, velocity and acceleration. Trick expects data to be in headers. Trick has a special way of parsing headers. The comment fields are used by Trick for units, IO specification and auto-documentation.

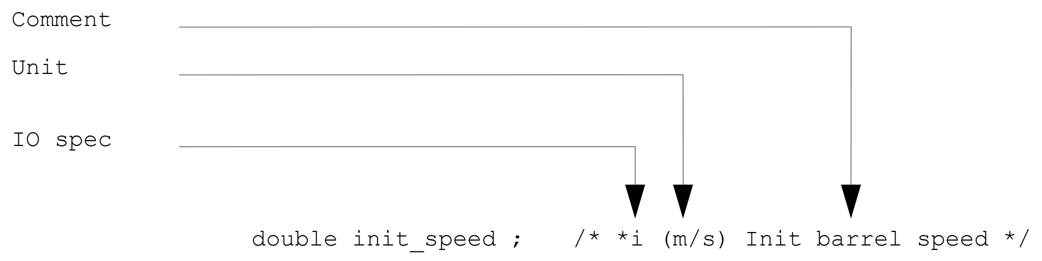
**Figure 3 Cannonball Header 1 – cannon.h**

```
/********************************* TRICK HEADER *************************
PURPOSE:     (my first cannonball test)
*************************************************************************/
#ifndef _cannon_h_
#define _cannon_h_

typedef struct {

       double pos0[2] ;   /* *i (m) Init position of cannonball */
       double vel0[2] ;   /* *i (m/s) Init velocity of cannonball */
       double acc0[2] ;   /* *i (m/s2) Init acceleration of cannonball */

       double pos[2] ;   /*  (m) xy-position */
       double vel[2] ;   /*  (m/s) xy-velocity */
       double acc[2] ;   /*  (m/s2) xy-acceleration  */
       double init_speed ; /* *i (m/s) Init barrel speed */
       double init_angle ; /* *i (r) Angle of cannon */

} CANNON ;

#endif
```

Some important notes:

- The "PURPOSE:" field is the keyword that Trick looks for in the header to identify the file as a Trick file so that Trick will know it needs to do some work. There are other optional fields. We'll leave those out for now. Note that the syntax requires a keyword colon followed by a definition in parenthesis.
- Trick understands structs, typedefs, enumerations, C++ classes, etc. We will stick with typedef structs for the time being. The typedef struct was the original technique used, and it is still the most popular among developers.
- Notice the comments next to each declaration. Comments to the immediate right of a variable declaration in a header file are assumed to be Trick comments. They deserve a little explanation. See Figure 4 and the section below. Developers may insert their own comments between the lines of declaration.

**Figure 4 Comment Field Next To Variable Declaration**



### 3.4.2.1 IO Specification

This is an optional field which defaults to *io if left unspecified. There are four options for this field. If you are a purist when it comes to data flow, you may use these parameters frequently.

- *i tells Trick that the parameter will be input only.
- *o tells Trick that the parameter is output only.
- *io is the default and means the parameter may be used for input or output.
- ** tells Trick that you do NOT want this parameter processed.

### 3.4.2.2 Unit Specification

Trick allows a unit specification so that it can conveniently do conversions for you during input file processing and when creating pretty plots. The unit specification is also used in the auto-documentation.

If the variable you are putting a comment next to is unit-less, use "--" as a unit.

Refer to the Trick user's guide for a complete list of units that Trick understands. To see a summary of the list, issue the following command (this is subject to change!!!):

**UNIX Prompt>** `ICG_10 -u`

### 3.4.2.3 Comment

The comment field is optional and is used in the auto documentation.

## 3.4.3 Creating The Trick Header

How do we create this header? Choose your favorite editor and type it in! You might get away with a cut-n-paste from Figure 3. The cut-n-paste technique is faster, but there is some concern that you may blaze pass and miss subtle issues. The tried-and-true way is to bite the bullet and do it from scratch. Doing it from scratch normally creates errors which you can learn from. May guilt follow you if you cut-n-paste :-)

We'll assume from this point on that your favorite editor is "vi". So when you see "vi" just replace it with "emacs", "nedit", "jot", "wordpad", "kate", "bbedit", etc.

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/include`
**UNIX Prompt>** `vi cannon.h <edit and save the file>`

### 3.4.4 Default Data

In the Trickless simulation, variables were declared. After declarations, the variables were initialized. In the previous sections (3.4.2- 3.4.3), variables were declared in a structure. In this section, default values will be assigned to the variables. There are multiple methods to instruct Trick what default values to assign to the variables in the structure (input file, “default data” file, or “default data” job). The method used here will be the “default data” job (more information of default data jobs in section 3.5.5).

**Figure 5 Cannonball Default Data File**

```
/****************************** TRICK HEADER ***************************
PURPOSE: (Set the default data values)
 **********************************************************************/

#include "../include/cannon.h"

int cannon_default_data(CANNON* C)
{
   const double PI = 3.141592 ;

   C->pos[0] = 0.0 ;
   C->pos[1] = 0.0 ;
   C->acc[0] = 0.0 ;
   C->acc[1] = -9.81 ;
   C->init_angle = PI/6 ;
   C->init_speed  = 50.0 ;

   return(0) ;
}
```

There are some interesting features to note.

- This is just a C function. Trick will use a C compiler, gcc if you are lucky, to create an object out of this routine. Eventually, through Trick parsing, code generation & compilation (i.e., Trick voodoo), the object will be linked into a simulation executable. Since this is C code, you have the full power of the C language. You may make full use of any external libraries, whether system libraries or your own libraries.
- The "PURPOSE:" tag is in the comment. It needs to be there. Again there are many options for this special Trick comment, but for the sake of simplicity, we’ll leave them out for now.
- The CANNON structure referenced in the argument list is the typedef found in cannon.h. Trick needs to know what structure the variable is associated with so it can assign the default value to the appropriate variable.
- If a routine returns an int, which is not mandatory, Trick will use the value of the returned integer. If the routine returns a negative value, the simulation will terminate itself, otherwise normal operations occur.

```
UNIX Prompt>  cd $HOME/trick_models/cannon/gravity/src
UNIX Prompt>  vi cannon_default_data.c <edit and save>
```

### 3.4.5 Cannonball Initialization Source - cannon_init.c

Data declarations are now complete. Default values have been assigned to the declared variables. This brings us to routines that initialize data.

**Figure 6 Cannonball Initialization – cannon_init.c**

```
/********************************** TRICK HEADER *************************
PURPOSE: (Initialize the cannonball)
*************************************************************************/
#include <math.h>
#include "../include/cannon.h"

int cannon_init( CANNON* C)
{
       C->pos0[0] = C->pos[0] ;
       C->pos0[1] = C->pos[1] ;

       C->vel[0] = C->init_speed*cos(C->init_angle);
       C->vel[1] = C->init_speed*sin(C->init_angle);
       C->vel0[0] = C->vel[0] ;
       C->vel0[1] = C->vel[1] ;

       C->acc0[0] = C->acc[0] ;
       C->acc0[1] = C->acc[1] ;

       return(0) ;
}
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`

**UNIX Prompt>** `vi cannon_init.c <edit and save>`

## 3.4.6 Cannonball Model Source – cannon_analytic.c

The data has been declared and initialized. It is time to use the data! For instructive purposes, the most obvious route will be taken. Assume, given a time stamp, that the position of the ball is known. In this particular case, the position is known for each time stamp (i.e., there is an analytical solution).

**Figure 7 Cannonball Source – cannon_analytic.c**

```
/*********************************** TRICK HEADER **************************
PURPOSE:    ( Analytical Cannon )
****************************************************************************/
#include <stdio.h>
#include "../include/cannon.h"

int cannon_analytic(CANNON* C)
{
   static double time = 0.0 ;

   C->acc[0] = 0.0 ;
   C->acc[1] = -9.81 ;

   C->vel[0] = C->vel0[0] + C->acc0[0]*time ;
   C->vel[1] = C->vel0[1] + C->acc0[1]*time ;

   C->pos[0] = C->pos0[0] + C->vel0[0]*time + (0.5)*C->acc0[0]*time*time ;
   C->pos[1] = C->pos0[1] + C->vel0[1]*time + (0.5)*C->acc0[1]*time*time ;

   /*
    * Increment time by the time delta associated with this job
    * Note that the 0.01 matches the frequency of this job
    * as specified in the S_define.
    */
   time += 0.01 ;

   return(0);
}
```

This routine looks much like the routine found in our Trickless simulation. It is the piece that was surrounded by a while() loop. It is part of the "executive" functioning of the simulation. Underneath, Trick will surround this job with its own while() loop. As in the case of the cannon_init() routine, there is nothing particularly special about cannon_analytic(). It is just another C function that will be compiled into an object. Later, with much magic, the object will be linked into a simulation executable.

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`
**UNIX Prompt>** `vi cannon_analytic.c <edit and save>`

### 3.4.7 Cannonball Prototypes

We need a header file containing our prototypes for the Simulation Definition file (which will be described in section 3.5). This is only required for the C functions (not needed for C++ methods). Our simulation will not compile without this information.

**Figure 8 cannon_proto.h**

```
/*************************************************************
PURPOSE:     (Cannonball Prototypes)
*************************************************************/

#ifndef _cannon_proto_h_
#define _cannon_proto_h_

#include "cannon.h"

#ifdef __cplusplus
extern "C" {
#endif

int cannon_analytic(CANNON*) ;
int cannon_init(CANNON*) ;
int cannon_default_data(CANNON*);

#ifdef __cplusplus
}
#endif

#endif
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/include`

**UNIX Prompt>** `vi cannon_proto.h <edit and save>`

### 3.4.8 Cannonball Cleanup And Shutdown

We vote for skipping the shutdown routine. Save your fingers some typing!

## 3.5 Pulling It All Together With The Simulation Definition File

Disparate pieces of a simulation have now been created by you in the model "src" and "include" directories. In order for Trick to use this "model code" to create a simulation, a simulation definition file or "S_define" needs to be created.

**Figure 9 Simulation Definition - S_define**

```
/*************************** Trick Header ******************************
PURPOSE: (S_define Header)
LIBRARY_DEPENDENCY: ((cannon/gravity/src/cannon_analytic.c)
                     (cannon/gravity/src/cannon_init.c)
                     (cannon/gravity/src/cannon_default_data.c))
*********************************************************************/
#include "sim_objects/default_trick_sys.sm"

##include "cannon/gravity/include/cannon.h"
##include "cannon/gravity/include/cannon_proto.h"

class CannonSimObject : public Trick::SimObject {
   public:
       CANNON cannon ;
       CannonSimObject() {
           ("initialization") cannon_init( &cannon ) ;
           ("default_data") cannon_default_data( &cannon ) ;
           (0.01, "scheduled") cannon_analytic( &cannon ) ;
       }
} ;

CannonSimObject dyn ;
```

The S_define file syntax is C++ with a couple of Trick specific constructs. This S_define is composed of one "sim_object": dyn. Let's take the "dyn" object apart piece by piece. The name "dyn" is arbitrary. We could have called it "mickey_mouse".

### 3.5.1 Trick Header

```
/*************************** Trick Header *******************************
PURPOSE: (S_define Header)
LIBRARY_DEPENDENCY: ((cannon/gravity/src/cannon_analytic.c)
                     (cannon/gravity/src/cannon_init.c)
                     (cannon/gravity/src/cannon_default_data.c))
***********************************************************************/
```

- `PURPOSE: (S_define Header)`
  This tells Trick there is a header to process.

- `LIBRARY_DEPENDENCY: ((cannon/gravity/src/cannon_analytic.c)…`
  This line will list all of your c routines in parenthesis so Trick will know which files to process and where to find the files. Trick uses your $TRICK_CFLAGS environment variable (see section 3.6.1) in conjunction with "cannon/gravity/src" to find the listed files. The entire path name following the $TRICK_CFLAGS path must be included.

### 3.5.2 Included Files

- `#include "sim_objects/default_trick_sys.sm"`
  This line is mandatory for Trick specific functionality which you do not need to know about now.

- `##include "cannon/gravity/include/cannon.h"`
  All classes or structures that are referenced in the S_define file must have the file that the class/structure is defined in listed with a "##include" statement.

- `##include "cannon/gravity/include/cannon_proto.h".`

The above ##include file was created in 3.4.7. It contains the prototypes of all of the C jobs. It is required to allow C++ access to the C functions referenced in the S_define file. You may also put the prototypes in the S_define block as shown below. But if you need to call any of the C functions from the input file then you must include the prototypes in a header file (which is the preferred method).

**Figure 10 Example putting prototypes in S_define**

```
/*************************** Trick Header *******************************
PURPOSE: (S_define Header)
LIBRARY_DEPENDENCY: ((cannon/gravity/src/cannon_analytic.c)
                     (cannon/gravity/src/cannon_init.c)
                     (cannon/gravity/src/cannon_default_data.c))
***************************************************************************
/
#include "sim_objects/default_trick_sys.sm"

##include "cannon/gravity/include/cannon.h"

%{
extern "C" {
   extern int cannon_analytic(CANNON*) ;
   extern int cannon_init(CANNON*) ;
   extern int cannon_default_data(CANNON*);
}
%}
       .
       .
```

## 3.5.3 Data Lines

`Class CannonSimObject : public Trick::SimObject {`
The “sim_object” is defined as a C++ class and must be derived from the base class SimObject.

- `Class CannonSimObject`
  The name of the sim_object class is arbitrary.
- `public Trick::SimObject`
  As mentioned above, your sim_object class must be derived from the Trick base class SimObject.

`public : CANNON cannon ;`

- `CANNON`
  This is the name of the structure typedef that you created in the cannon.h header.
- `cannon`
  This is an alias for the CANNON structure. It is mandatory.
- `CannonSimObject() {`
  This is the constructor of the sim_object and it will contain the job declarations.

## 3.5.4 Initialization Job

It is custom to refer to the routines or C-functions created by the developer as "jobs". The statement below tells Trick how to handle the cannon_init() job.

```
("initialization") cannon_init( &cannon) ;
```

- `("initialization")`
  This assigns cannon_init() a job classification of "initialization". There are many classes of jobs. The job classes, for the most part, determine the order the job is called in the "executive loop". If there are two jobs of the same class in the S_define, the one seen first in the S_define is called first. Jobs that are classified "initialization" will be called once before the main executive loop and will not be called again.

- `cannon_init(`
  The name of the function we created in $HOME/trick_models/cannon/gravity/src/cannon_init.c.

- `&cannon)`
  This is the actual value passed to cannon_init(). It is the address of the object's cannon structure and "cannon" is the alias for the CANNON structure.

### 3.5.5 Default Data Job

The default data jobs are called one time before the initialization jobs.

```
("default_data") cannon_default_data(&cannon) ;
```

- `("default_data")`
  This assigns cannon_default_data() a job classification of "default_data".

### 3.5.6 Scheduled Job

The next job needs to be called at a regular frequency while the cannonball is flying in the air. A "scheduled" class job is one of many jobs that can be called at a given frequency. The only thing new about the declaration for cannon_analytic() is the additional specification of 0.01.

```
(0.01, "scheduled") cannon_analytic(&cannon) ;
```

- `(0.01, "scheduled")`
  The 0.01 specification tells Trick to run this job every 0.01 seconds (starting at time=0.0). The keyword "scheduled" is the job's classification.

### 3.5.7 Create The S_define

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`
**UNIX Prompt>** `vi S_define <edit and save, see Figure 9>`

## 3.6 Building The Simulation

The pieces are in order. The simulation is ready to be built!

### 3.6.1 Setting $TRICK_CFLAGS -- Don't Even Skip This Section!!!

BEFORE we continue with the magical building of the cannonball, PLEASE take a second to understand this section. It will save you much heartache and time.

Trick cannot find your structures, your headers or your functions without a little direction.

Trick uses the -I paths in your $TRICK_CFLAGS environment variable to find your structures, headers and functions. The -I paths are called "include" paths.

In the S_define, relative paths (e.g., cannon/gravity) were specified. Full paths (e.g., /home/my_name/trick_models/cannon/gravity/include) were not specified. Trick must have some way of making a full path from the relative path in the S_define. It pre-pends the paths given in the -I's of the $TRICK_CFLAGS to make full paths from the relative paths.

There are many ways to setup your TRICK_CFLAGS. One way is to make a new file called ".Trick_user_cshrc", if you are using the csh or tcsh environment.

```
UNIX Prompt>  cd $HOME
UNIX Prompt>  vi .Trick_user_cshrc
  Add line:
     setenv TRICK_CFLAGS "$TRICK_CFLAGS -I$HOME/trick_models"
UNIX Prompt>  source ~/.cshrc
  .cshrc will source .Trick_cshrc which sources .Trick_user_cshrc
  which gives a new environment with TRICK_CFLAGS set appropriately
```

In the bash environment, do the following:

```
UNIX Prompt>  cd $HOME
UNIX Prompt>  vi .Trick_user_profile
  Add lines:
     TRICK_CFLAGS="$TRICK_CFLAGS -I$HOME/trick_models"
     export TRICK_CFLAGS
UNIX Prompt>  . .profile
```

Having set up TRICK_CFLAGS, verify with the following:

```
UNIX Prompt>  gte TRICK_CFLAGS
  Verify that -I$HOME/trick_models is in your $TRICK_CFLAGS
  gte stands for "Get Trick Environment"
```

**- OR -**

Create a file named S_overrides.mk in your simulation directory and add the following line to it:

```
TRICK_CFLAGS += -I${HOME}/trick_models
TRICK_CXXFLAGS += -I${HOME}/trick_models
```

### 3.6.2 CP

The source code and environment are set up. Trick's main processor is called CP (Configuration Processor). It is responsible for parsing through the S_define, finding structures, functions, and ultimately creating a simulation executable. Give it a whirl.

```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_analytic
UNIX Prompt>  CP
```

If you typed everything perfectly... Trick is installed properly... there are no bugs in the tutorial... the stars are aligned... and Trick is in a good mood... You just might have a simulation.

```
UNIX Prompt>  ls
   Is there an S_main*exe file??? If so, cool deal.  If not, scream!
```

If all went well, you will notice several other files now resident in the SIM_cannon_analytic directory. S_source.cpp is an interesting auto-generated C++ source file. It should thoroughly confuse anybody!

### 3.6.2.1 A Perfect Build

Under good conditions you should have seen output similar to the following. We've selectively taken out some of the messages and changed others to make it fit on one page.

**Figure 11 CP Output (Abridged)**

```
Parsing S_define...
       :
Continuing S_define
Processing sim_class CannonSimObject

Completed parsing S_define

Creating S_source.c... Complete
Determining structure dependencies.
ICG'ing header files to get all header file dependencies...
ICG /users/dcvillar/trick_dev/trunk/trick_sims/SIM_cannon_analytic/S_source.hh
       :
Getting dependencies for
/users/dcvillar/trick_dev/trunk/trick_sims/SIM_cannon_analytic/S_source.hh

All header file dependencies found.
Determining module dependencies.
Getting dependencies for
/users/dcvillar/trick_dev/trunk/trick_models/cannon/gravity/src/cannon_init.c
Getting dependencies for
/users/dcvillar/trick_dev/trunk/trick_models/cannon/gravity/src/cannon_analytic.c
Getting dependencies for
/users/dcvillar/trick_dev/trunk/trick_models/cannon/gravity/src/cannon_default_data.c

       :
/usr/bin/c++ -Wl,--export-dynamic -o
/users/dcvillar/trick_dev/trunk/trick_sims/SIM_cannon_analytic/S_main_Linux_4.1_x86_64.exe \
       :
=== Simulation make complete ===
```

### 3.6.2.2 Troubleshooting A Bad Build

Here are some common problems.

- Trick cannot seem to find a function or structure that you have in your S_define.
  - Make sure that your TRICK_CFLAGS are set. See section 3.6.1.
  - You have a misspelling.
  - In order for Trick to find a job, argument types must match exactly.

- Trick barfs when building the simulation
  - One of your C routines may not compile because of a C syntax error.
  - Trick was not installed properly on your system.

- CP croaks
  - You may have a syntax error in your S_define.

## 3.7 Running The Simulation

You've done a lot of work to get to this point. You've created a header, a default data job, an initialization job, a scheduled job, and an S_define. You've also had to set up an environment and trudge through CP errors. The tiny Trickless main() program may be looking short-n-sweet at this point! There can't be anything more to do!?!

There is one more file to create to get the cannonball out of the barrel and into the air.

### 3.7.1 Simulation Input File

Every Trick simulation needs an input file. This input file will be simple (only one line). In practice, input files can get ridiculously complex. The input file is processed by Python. There is no need to recompile the simulation after changing the input file. The file is interpreted.

**Figure 12 Input File – input.py**

```
trick.stop(5.2)
```

By convention, the input file is placed in a RUN_* directory.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`
**UNIX Prompt>** `mkdir RUN_test`
**UNIX Prompt>** `cd RUN_test`
**UNIX Prompt>** `vi input.py <edit and save>`

### 3.7.2 Execute S_main*.exe

To run the simulation, simply execute the S_main*exe.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`
**UNIX Prompt>** `./S_main_*.exe RUN_test/input.py`

If all is well, something similar to the following sample output will be splashed on the terminal.

**Figure 13 S_main*exe Message**

```
|L   0|2012/05/29,09:54:27|WonderWoman| |T 0|5.200000|
SIMULATION TERMINATED IN
  PROCESS: 0
  ROUTINE: Executive_loop_single_thread.cpp:91
  DIAGNOSTIC: Reached termination time

       SIMULATION START TIME:      0.000
        SIMULATION STOP TIME:      5.200
    SIMULATION ELAPSED TIME:      5.200
        ACTUAL CPU TIME USED:      0.090
      SIMULATION / CPU TIME:     57.786
    INITIALIZATION CPU TIME:      0.086
```

### 3.7.3 What Happened?

Now what? The simulation has run? What did it do? In the immortal words of Elmer Fudd, "Where did he go?" Why was it so fast? Isn't Trick "real time"? Where is the trajectory?

To get a peek into the simulation, you may, among other things, record data and stripchart data while running real-time.

## 3.7.4 Recording Data

### 3.7.4.1 Creating A Data Recording File

Recording data is possible when running real-time and non-real-time (as fast as the computer will run). In order to record data, it is necessary to create a "data recording file". You may create the file using a GUI called dre (data recording editor --- aka Dr. Dre) or you may create it manually.

### 3.7.4.2 Creating a Data Recording File Using Dre

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`
**UNIX Prompt>** `mkdir Modified_data`
**UNIX Prompt>** `dre &`

Step 1. In the `"DR Name"` entry box, enter `my_cannon`.

Step 2. In the `"DR Cycle"` entry box, change `0.1` to `0.01`.

Step 3. In the `"Variables"` pane, *double-click* `dyn`, then *double-click* `cannon`.

Step 4. *Double-click* `pos[2]`. `dyn.cannon.pos[0]` and `dyn.cannon.pos[1]` should appear in the `"Selected Variables"` pane.

Step 5. Choose `File->Save`.
In the `"Save"` dialog, enter the file name `cannon.dr`.
Save `cannon.dr` in the `Modified_data` directory.

Step 6. Exit dre.

Storing the data recording file `"cannon.dr"` into a directory called `"Modified_data"` is not mandatory. It is just common convention. You may view `"cannon.dr"` it is a text file.

### 3.7.4.3 Creating a Data Recording File Manually

If you created the data recording file using the steps in the previous section, then skip this section.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`
**UNIX Prompt>** `mkdir Modified_data`
**UNIX Prompt>** `vi Modified_data/cannon.dr (edit and save, see Figure 14)`

**Figure 14 cannon.dr**

```
drg = trick.DRBinary("my_cannon")

drg.add_variable("dyn.cannon.pos[0]")
drg.add_variable("dyn.cannon.pos[1]")

drg.set_cycle(0.01)
drg.set_freq(trick.DR_Always)

trick.add_data_record_group(drg,trick.DR_Buffer)
drg.enable()
```

#### 3.7.4.4 Running The Simulation And Recording Data

The simulation must know about the data recording file created in the last section. This is accomplished by adding execfile to the simulation input file.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic/RUN_test`

**UNIX Prompt>** `vi input.py`
```
   Add the line: execfile("Modified_data/cannon.dr")
```

**UNIX Prompt>** `cd ..`

**UNIX Prompt>** `./S_main*.exe RUN_test/input.py`

After the simulation runs, data will be dumped into the RUN_test directory. The data, in this case, is recorded in binary form.

## 3.8 Viewing Data

To view the data which was recorded, another hill needs climbing. Trick's "data products" must be learned. In this section, the data will be viewed using an application called "quick plot". *Note: The GUI or plot graph figures may be shown differently on your machine due to the differences of the platforms and ongoing changes.*

### 3.8.1 Using Trick's Quick Plot

Begin by launching trick_dp (trick data products GUI).

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`

**UNIX Prompt>** `trick_dp &`

#### 3.8.1.1 Plotting Time -vs- Position

Step 1. *Double click* the pathname containing your sim directory (or single click the symbol next to the name)

Step 2. *Double click* the `SIM_cannon_analytic` name in the `Sims/Runs` pane.
This will reveal the `RUN_test` directory.

Step 3. *Double click* the `RUN_test` name.
This will bring `RUN_test` into the `RUN Selections` pane below.

Step 4. *Click* the blue lightning button in the tool bar to launch quick plot (qp).
The qp GUI will pop up.

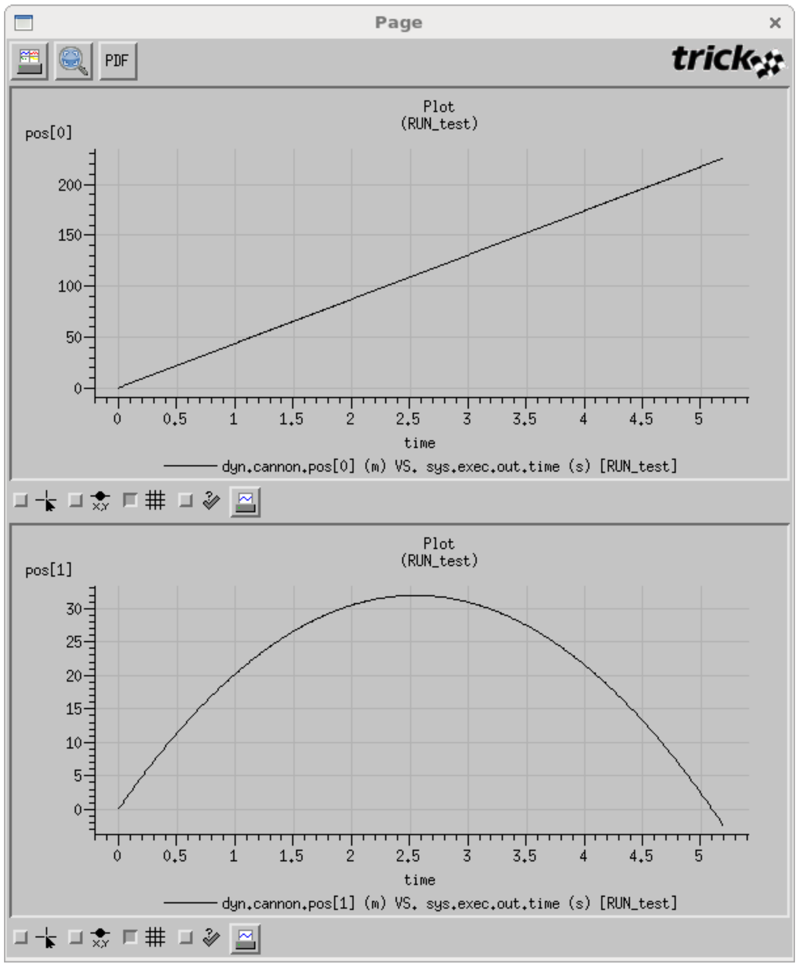
Step 5. In qp, *right click* the `dyn.cannon.pos[0-1](m)` variable in the left pane and choose expand var.
Next *double click* `dyn.cannon.pos[0](m)`.
This sets up qp to create one page with one plot (`time` -vs- `pos[0]`).

Step 6. Now *click* the `dyn.cannon.pos[1]` variable and *drag* it to the pane on the right. *Drop* it on the line with `"Page"` (see the white-n-black window looking icon).
This will result in one page containing two plots.

Step 7. In qp, *click* the plain white sheet icon located on the toolbar.
A single window with two plots should pop up. (See Figure 15)

Step 8. When you are done with the plots you created, close the fxplot window which will also close the window with your plot(s).

**Figure 15 Quick Plot Time -vs- Position**


Page
PDF
trick
Plot
(RUN_test)
pos[0]
200
150
100
50
0
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
time
dyn.cannon.pos[0] (m) VS. sys.exec.out.time (s) [RUN_test]
Plot
(RUN_test)
pos[1]
30
25
20
15
10
5
0
time
dyn.cannon.pos[1] (m) VS. sys.exec.out.time (s) [RUN_test]

### 3.8.1.2 Plotting XPosition -vs- YPosition

Step 1. There are a couple of options to clear the plots from the “DP Content” pane.

a. *Right click* `"Plots"` in the `"DP Content"` pane and *choose* `"Remove All Pages"`.

b. *Click* the `"New"` plot icon located on the far left of the toolbar. *Click* `"Ok"` when asked if you want to start over.

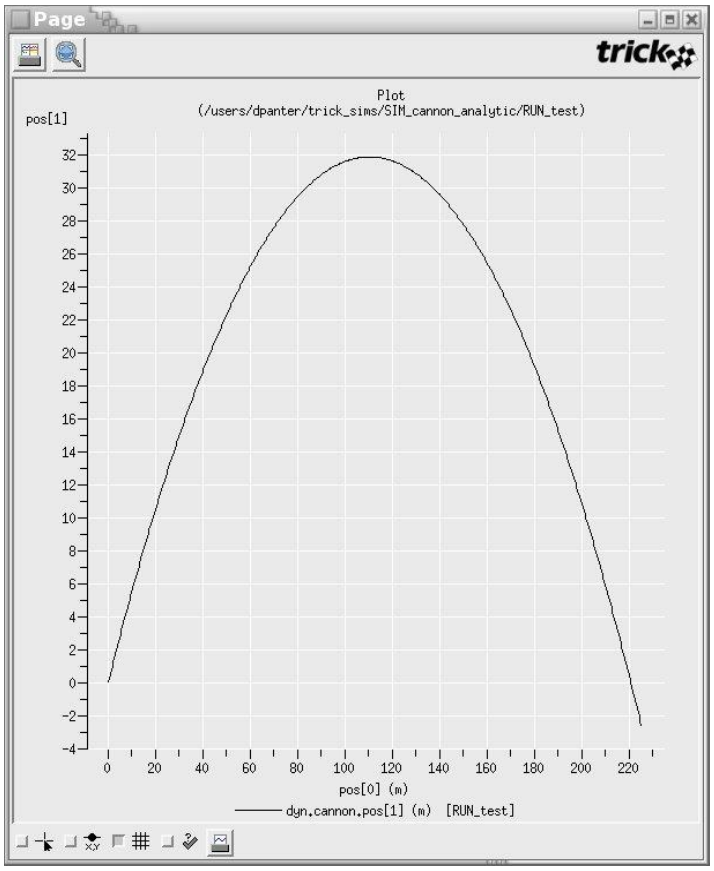
*Step 2.* *Double click* `dyn.cannon.pos[1]`.

Step 3. *Drag-n-drop* the `dyn.cannon.pos[0]` variable over the `sys.exec.out.time` variable in the `Plot` located in the `"DP Content"` pane. You will be asked to confirm the replacement. *Click* `"Ok"`.

Step 4. To see the plot, *click* the white sheet icon on the toolbar.
Voila! There is the trajectory! :-)
Note that the cannonball goes underground. This will be fixed later.

Step 5. When you are done, close the fxplot window.

**Figure 16 Quick Plot X -vs- Y Position**



### 3.8.1.3 Creating a DP Specification File

It would get old fast clicking these variables over and over again using the quick plot. To get around this, the information needed for this plot will be saved off to a file named `DP_cannon_xy`, and then reused by trick_dp. This is an important step; extensive use will be made from the `DP_cannon_xy` file.

Step 1. With the qp GUI still up and the x -vs- y position still chosen, *click* the `dyn.cannon.pos[1]` variable located in the pane on the right (in the `DP Content pane`). The `dyn.cannon.pos[1]` variable should be highlighted. The “`Y Var`" notebook page should be visible in the lower right pane.

Step 2. In the "`Y Var`" notebook page, *select* "`Symbol Style->Circle`" from the drop-down menu.

Step 3. In the "`Y Var`" notebook page, *select* "`Symbol Size->Tiny`" from the drop-down menu.

Step 4. *Click* the "`Apply Change`" button (you may have to scroll up to see the button).

Step 5. *Save* this information by clicking the menu option "`File->Save As`". *Click* "`New Folder`" button to create the DP_Product folder.
Choose the directory button "`SIM_cannon_analytic/DP_Product`". Enter file name as "`DP_cannon_xy`".

Step 6. Close the quick plot GUI, but keep trick_dp up and running.

### 3.8.1.4 Using trick_dp To View Data

Now that `DP_cannon_xy` has been saved, the data can be viewed with trick_dp.

Step 1. Assuming the trick_dp is still up and running from the previous steps, *click* "`Session->Refresh…`" to reveal `DP_cannon_xy.xml` in the top right pane, "`DP Tree`".

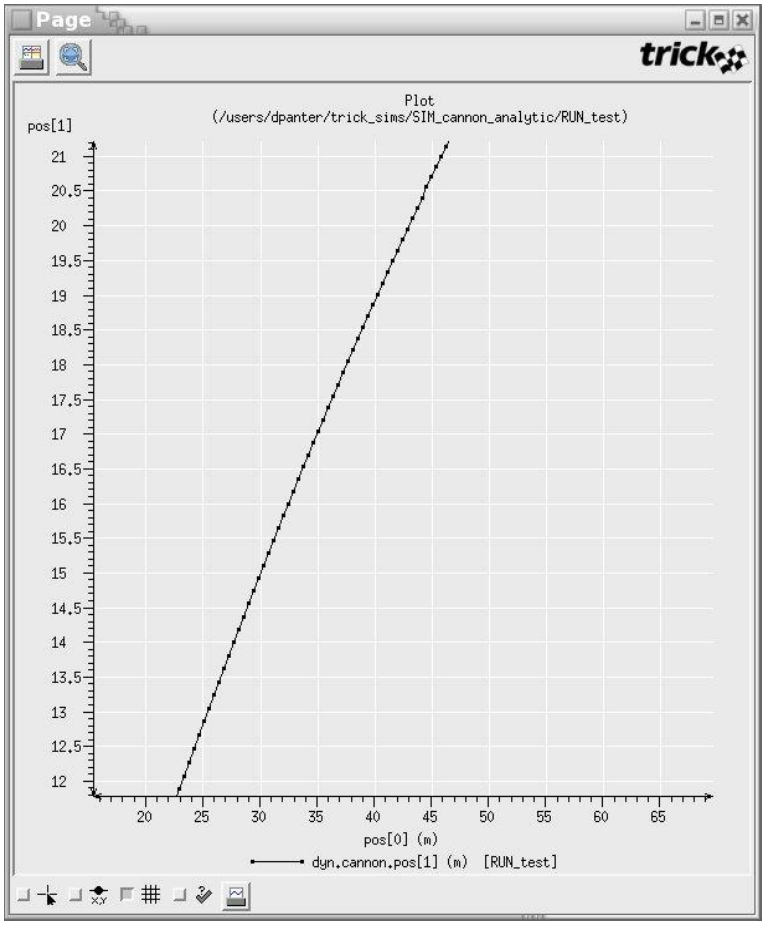
Step 2. Make sure that "`Sims/Runs->SIM_cannon_analytic/RUN_test`" shows up in the "`Run Selections`" pane. If not, then double click it to add it.

Step 3. Choose the "`DP_cannon_xy.xml`" in the top right pane by double clicking it.
This will bring the "`DP_cannon_xy.xml`" into the "`DP Selections`" pane.

Step 4. To see the trajectory again, *click* the plain white single sheet button on the toolbar.
Zoom in by holding the middle mouse button and drag across a section of the plot. Then release the mouse button. Notice that there is a tiny circle on each x-y point recorded.

Step 5. Once you are finished with the plot, close the fxplot window and Trick DP.

**Figure 17 Zoomed X -vs- Y Position - DP_cannon_xy**



## 3.9 Running Real-Time

Recall that the cannonball run was 5.2 seconds, yet when the simulation ran it was done in a flash of CPU time. This section will add real-time synchronization.

### 3.9.1 Making A Real-time Input File

Similar to making a data recording file, a real-time input file is needed.

**Figure 18 Real Time File - realtime.py**

```
trick.frame_log_on()
trick.real_time_enable()
trick.exec_set_software_frame(0.1)
trick.itimer_enable()
trick.exec_set_enable_freeze(True)
trick.exec_set_freeze_command(True)

trick.sim_control_panel_set_enabled(True)
```

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic/Modified_data`
**UNIX Prompt>** `vi realtime.py <edit and save>`

The settings in this realtime.py are a little obscure. Here is a brief explanation.

*`trick.frame_log_on()`* - tells the simulation to log the performance of the simulation (i.e., how it is running with respect to the real-world time).

*`trick.real_time_enable()`* - tells the simulation to run in real-time synchronization.

*`trick.exec_set_software_frame(0.1)`* - tells the simulation the frequency of the "heartbeat" of the simulation. If the simulation were sending out packets of information to a graphics server, the "heartbeat" might be 50 HZ (0.02). Trick synchronizes with the system clock on multiples of the rt_software_frame. If it is beating the system clock, it pauses. If it is falling behind, it registers "timeouts" and attempts to catch up.

*`trick.itimer_enable()`* - configures the simulation to use system interval timer signals as the heartbeat of the simulation. It allows other processes to run while Trick is waiting for the beginning of the next software frame to start the simulation's jobs. If interval timers are not used, Trick will spin waiting for the next beat.

*`trick.exec_set_freeze_command()`* - brings up the simulation in a frozen (non-running) state.

*`trick.exec_set_enable_freeze()`* - allows the user to toggle the simulation from a frozen state to a running state at will.

*`trick.sim_control_panel_set_enabled(True) or`*

*`simControlPanel = trick.SimControlPanel() &`*
*`trick.add_external_application(simControlPanel)`* - brings up the simulation control panel GUI.

The realtime.py file must be included in the RUN_test/input.py file. When finished, the latest version of the input file should look like the following:

**Figure 19 Input File With Real-time – input.py**

```
execfile("Modified_data/realtime.py")
execfile("Modified_data/cannon.dr")

trick.stop(5.2)
```

```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_analytic/RUN_test
UNIX Prompt>  vi input.py <edit and save>
```

### 3.9.2 Using The Sim Control Panel

Fire up the cannonball again.

```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_analytic
UNIX Prompt>  ./S_main*.exe RUN_test/input.py &
```

The "simulation control panel" should popup now. In the realtime.py file, we instructed Trick to start in a "freeze" state. Currently, the ball is sitting in the base of the barrel. To fire it, the "Start" button must be clicked on the simulation control panel in the "Commands" box.

Step 1. *Click* `"Start"` on simulation control panel.
The simulation will run in sync with real-time.

Step 2. Once the simulation is finished, *click* the `"Exit"` button.

Some items to note about the simulation control panel for your future use:

- You may "freeze" the simulation at any point, and then restart.
- You may freeze the simulation, then "dump a checkpoint" of the current state. This state is reloadable (i.e., you may jump to a previous/future point in the sim by loading the appropriate checkpoint).
- You may toggle between real-time and non-real-time.
- If the simulation lags behind real-time, it will log these as overruns (does not complete all jobs during the software frame) and display them in the tiny box next to the simulation name. If the simulation overruns, the sim will run as fast as it can "to catch up" to where it should be.
- Using the File menu at the top, you may set a freeze point in the future.

## 3.10 Viewing Real-Time Data

We saw in section 3.8 how to view post-run data. Trick offers two ways to view data as the simulation is running. You may stripchart the run-time data or view the run-time data through a special application called TV. TV stands for "Trick View" (for history's sake, the acronym TV preceded the name Trick View).

### 3.10.1 Stripcharting

There are two ways to stripchart. You may choose a variable on-the-fly or you may create a stripchart input file. We will start by using an input file.

**Figure 20 Stripchart Input File - cannon.sc**

```
Stripchart:
   title = "Cannon Trajectory"
   geometry = 800x800+300+0
   x_min = 0.0
   x_max = 250.0
   y_min = 0.0
   y_max = 40.0
   x_variable = dyn.cannon.pos[0]
   dyn.cannon.pos[1]
```

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`
**UNIX Prompt>** `vi cannon.sc <edit and save>`

A brief explanation of the settings in the cannon.sc file:

`title`*:* The top of the window display will contain this string.

`geometry`*:* What size and where to place the stripchart.

*`xy min max`:* Since this is a stripchart, normally the stripchart will resize itself as data comes. In this case, the stripchart's data limits are predetermined with x_min, x_max, y_min and y_max.

*`x_variable`:* If not specified, it is "sys.exec.out.time".

*`dyn.cannon.pos[1]`:* The y variable to plot. There may be multiple curves per stripchart.

Trick must know about the newly created stripchart input file. Once again, this needs to be added to the RUN_test/input.py file. Once completed the new RUN_test/input file should look like the following. Since a stripchart will be popping up, the size of the "sim control panel" is reduced as well.

**Figure 21 Simulation Stripchart Input File – input.py**

```
execfile("Modified_data/realtime.py")
execfile("Modified_data/cannon.dr")

trick_vs.stripchart.set_input_file("cannon.sc")

trick.stop(5.2)
```

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic/RUN_test`
**UNIX Prompt>** `vi input.py <edit and save>`

Run the simulation, and if all is beautiful, you will see a real-time trajectory of the cannonball.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`
**UNIX Prompt>** `./S_main*exe RUN_test/input.py &`

Once it is finished, exit the sim control panel and close the stripchart.

### 3.10.2 TV - Trick View

Another application for viewing/setting parameters is called Trick View (TV). TV may be launched with an input file, or may be launched directly from the sim control panel. You may also leave TV running between simulation runs. You will need to click the "`Connect`" button in the lower right corner of the TV GUI to re-connect to the simulation.

#### 3.10.2.1 TV Without An Input File

Step 1. Fire up the simulation

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`

**UNIX Prompt>** `./S_main*exe RUN_test/input.py &`

Step 2. *Start* TV by either *clicking* the blue TV icon button or *choosing* sim control panel's menu option: "`Actions->Start TrickView`".
The TV GUI should pop up.

Step 3. In the "Variables" pane, *double-click* `dyn`, and then *double-click* `cannon`.

Step 4. Pick a few variables of interest to view by double-clicking on the variables, for instance "`dyn->cannon->init_angle`","`dyn->cannon->pos`", "`dyn->cannon->init_speed`". The variables will show up on the "Parameter" pane on the right.

Step 5. Since we have gone through the trouble of selecting position, init_speed and init_angle, let's go ahead and save these selections.

- `Click` the "Save" button on the TV toolbar, or select "`File->Save`" from the main menu.
- Save to the `SIM_cannon_analytic` directory, and name the file "`TV_cannon`".

Step 6. On the sim control panel, choose "`Actions->FreezeAt`". Enter in a time of 2.0 seconds. *Click* OK.

Step 7. On the sim control panel, *click* the "Start" button to put the simulation into action.
Notice that TV's parameter values changed. The cannonball position is now at [86.17, 30.33]
The stripchart should reflect this.

Step 8. Let us force the Y position of the cannonball to drop to 10 meters. To do this, *click* on the `dyn.cannon.pos[1]` variable on the TV Parameter table. Replace 30.325... with 10.0 as a new position. Hit <Enter> to set the variable with the new value. The stripchart should show the drop from 30 meters to 10 meters.

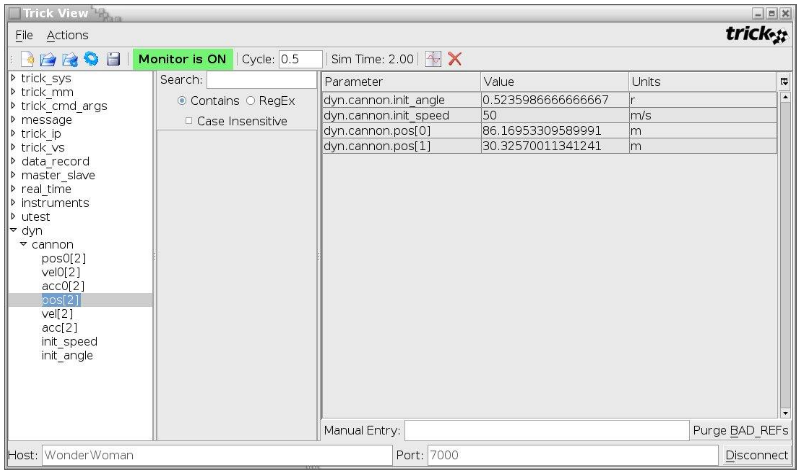
Step 9. With the `dyn.cannon.pos[1]` variable still highlighted on the TV Parameter table, *click* the "Stripchart" button on the toolbar. This starts a stripchart of this variable. Note that you may stripchart -any- variable. You may view/set any variable.

Step 10. Notice that the init_angle is 0.52359877... To view in degrees:

- *Click* on the variable `dyn.cannon.init_angle` on the Parameter table.
- *Click* on the "`r`" in the Unit column to bring up a drop-down list.
- *Select* "`d`", and notice that the value of `init_angle` changes to 30 degrees.

.

**Figure 22 Viewing/Setting Simulation in TV**



| Parameter | Value | Units |
|---|---|---|
| dyn.cannon.init_angle | 0.5235986666666667 | r |
| dyn.cannon.init_speed | 50 | m/s |
| dyn.cannon.pos[0] | 86.16953309589991 | m |
| dyn.cannon.pos[1] | 30.32570011341241 | m |

Step 11. Resume the simulation run by clicking the "Start" button on the sim control panel. Notice that the trajectory assumes its predetermined path. This is because we are giving the cannonball a position as a function of time.

## 3.10.2.2 TV With An Input File

If this simulation were run over and over, it would be laborious to clickety-click the variables each time. It would be advantageous to use the TV_cannon file we saved off in the last step. There are two ways to do this.

### 3.10.2.2.1 Loading A TV Input File Directly From TV

Step 1. Fire up the cannon.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_analytic`
**UNIX Prompt>** `S_main*exe RUN_test/input.py`

Step 2. *Choose* "`Actions->Start Trick View`" from sim control panel.

Step 3. *Click* the file "Open" or "Open" with a blue circle icon button on the TV toolbar or *select* `File->Open` or `File->Open & Set` from the main menu.

Step 4. *Select* the `TV_cannon` file saved off in the last run. The Parameter table on the right is cleared and replaced with all the saved variables.

Step 5. If "Set" is chosen for opeing a TV file, all saved values are restored as well. Otherwise, only variables are loaded to the Parameter table. If only "Set" is selected, corresponding vaiable values are restored without the Parameter table being changed.

This simulation is simple since we have a limited number of parameters. The streamlining is more pronounced when the simulation has thousands of variables, in which the process for selecting variables would become awfully repetitious.

### 3.10.2.2.2 Allowing The Simulation To Load A TV Input File

It even gets tiresome clicking the TV input file from TV. The following example shows how to configure the simulation to automatically launch the TV with the parameters of interest. The syntax for this file is similar to the stripchart input file.

Again, we need to incorporate the TV input file into our ever expanding simulation input file.

**Figure 23 Simulation TV Input File – input.py**

```
execfile("Modified_data/realtime.py")
execfile("Modified_data/cannon.dr")

trick_vs.stripchart.set_input_file("cannon.sc")

trick.trick_view_add_auto_load_file("TV_cannon.tv")

trick.stop(5.2)
```

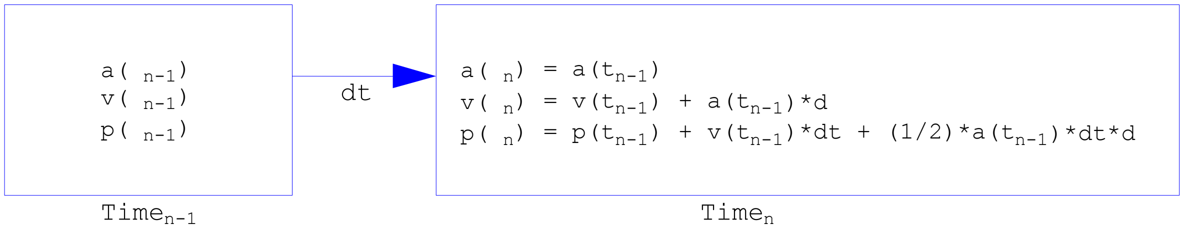
```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_analytic/RUN_test
UNIX Prompt>  vi input.py <edit and save>
```

You may now run the sim and verify that TV pops up automatically.

## 4.0 Dynamics - State Propagation Using dt

The model that we created in the last section is unique in that it has an analytical solution. This section will make another attempt at modeling the cannonball using a more generic technique. Instead of calculating position as a function of simulation time, we will predict the current state based on the previous state. We will calculate the current state at time$_n$ based on the state at time$_{n-1}$.

**Figure 24 Cannonball State Using Time Delta**


a( $_{n-1}$)
v( $_{n-1}$)
p( $_{n-1}$)
dt
a( $_n$) = a(t$_{n-1}$)
v( $_n$) = v(t$_{n-1}$) + a(t$_{n-1}$)*d
p( $_n$) = p(t$_{n-1}$) + v(t$_{n-1}$)*dt + (1/2)*a(t$_{n-1}$)*dt*d
Time$_{n-1}$
Time$_n$

This method will actually give the same answer as the analytic solution. However, the only reason it will yield the analytic solution is because our cannonball simulation happens to have constant acceleration.

## 4.1 Modifying Our Existing Cannonball Simulation

Instead of starting from scratch, we'll scam off of the previous cannonball simulation. The data, data initialization, and initialization routines will be identical. The only change will be in the scheduled job.

To separate the last cannonball simulation from this one, save it.

**UNIX Prompt>** `cd $HOME/trick_sims`
**UNIX Prompt>** `cp -r SIM_cannon_analytic SIM_cannon_dt`

### 4.1.1 Create Scheduled Job Using dt

**Figure 25 Cannonball Source 2 - cannon_dt.c**

```
/*********************************** TRICK HEADER *************************
PURPOSE:    ( Try dt )
***************************************************************************/
#include "../include/cannon.h"

int cannon_dt( CANNON* C )
{
       double dt ;
       double pos0[2] ;
       double vel0[2] ;
       double acc0[2] ;

       /* This dt matches the frequency of this job */
       dt = 0.01 ;

       /* Save off last state */
       pos0[0] = C->pos[0] ; pos0[1] = C->pos[1] ;
       vel0[0] = C->vel[0] ; vel0[1] = C->vel[1] ;
       acc0[0] = C->acc[0] ; acc0[1] = C->acc[1] ;

       /* Calculate new state based on last state */
       C->acc[0] = acc0[0] ;
       C->acc[1] = acc0[1] ;

       C->vel[0] = vel0[0] + acc0[0]*dt ;
       C->vel[1] = vel0[1] + acc0[1]*dt ;
       C->pos[0] = pos0[0] + vel0[0]*dt + (0.5)*acc0[0]*dt*dt ;
       C->pos[1] = pos0[1] + vel0[1]*dt + (0.5)*acc0[1]*dt*dt ;

       return(0) ;
}
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`
**UNIX Prompt>** `vi cannon_dt.c <edit and save>`

### 4.1.2 Modify cannon_proto.h

**Figure 26 cannon_proto.h**

```
/*******************************************************************
PURPOSE:           ( Try dt )
*******************************************************************/
#ifndef _cannon_proto_h_
#define _cannon_proto_h_

#include "cannon.h"

#ifdef __cplusplus
extern "C" {
#endif

int cannon_dt(CANNON*) ;
int cannon_init(CANNON*) ;
int cannon_default_data(CANNON*);

#ifdef __cplusplus
}
#endif
#endif
```

### 4.1.3 Modify S_define

The S_define call to "cannon_analytic()" must be modified to the newly created "cannon_dt()". Once done, the simulation can be rebuilt.

**Figure 27 Simulation Definition - S_define**

```
/*************************** Trick Header ********************************
PURPOSE: (S_define Header)
LIBRARY_DEPENDENCY: ((cannon/gravity/src/cannon_dt.c)
                     (cannon/gravity/src/cannon_init.c)
                     (cannon/gravity/src/cannon_default_data.c))
***************************************************************************
/
#include "sim_objects/default_trick_sys.sm"

##include "cannon/gravity/include/cannon.h"
##include "cannon/gravity/include/cannon_proto.h"

class CannonSimObject : public Trick::SimObject {
   public:
      CANNON cannon ;
      CannonSimObject() {
         ("initialization") cannon_init( &cannon ) ;
         ("default_data") cannon_default_data( &cannon ) ;
         (0.01, "scheduled") cannon_dt( &cannon ) ;
      }
} ;

CannonSimObject dyn ;
```

There are two small changes between the first S_define and this one. The changes are highlighted in the gray boxes. To make these changes follow the steps below.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_dt`
**UNIX Prompt>** `vi S_define <change cannon_analytic to cannon_dt>`

### 4.1.4 Compare Try One And Try Two

To compare the simulations we need to run SIM_cannon_dt.

Step 1. Build simulation
**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_dt`
**UNIX Prompt>** `make spotless`
**UNIX Prompt>** `CP`

Step 2. Turn off TV and turn off Stripcharting...
**UNIX Prompt>** `vi RUN_test/input.py`

```
Comment out the trick_view line by prepending #
Change: trick.trick_view_add_auto_load_file(“TV_cannon.tv”)
To:     #trick.trick_view_add_auto_load_file(“TV_cannon.tv”)
Comment out the stripchart line by prepending #
Change: trick_vs.stripchart.set_input_file(“cannon.sc”)
To:    #trick_vs.stripchart.set_input_file("cannon.sc")
```

Step 3. Run simulation
**UNIX Prompt>** `S_main*exe RUN_test/input.py &`

```
Since SIM_cannon_analytic was copied to SIM_cannon_dt, all the input
files are setup.  Run the simulation to completion.
```

Step 4. Use trick_dp to compare data responses of both sims
**UNIX Prompt>** `trick_dp &`

Step 5. Select `RUN_test` in the `SIM_cannon_analytic` folder

Step 6. Select `RUN_test` in the `SIM_cannon_dt` folder

Step 7. Select `DP_cannon_xy.xml` in the "DP Tree" pane from either of the above `SIM_` folders

Step 8. *Click* the toolbar Co-Plot button. The icon looks like collated white sheets.
The two plots are identical. We still have some work to do!

# 5.0 Dynamics - Trick Integration

The last method is decent in that it doesn't rely on the simulation being a function of time. It moves from one state to the next and does this quite well. There are some problems with it though.

- dt is hardcoded in the scheduled job.
- Most importantly, if acceleration were not constant, the response would not be correct. In fact, if the acceleration were not constant the previous method would be a poor way to propagate a state. Using the same technique with a spring would make a simulated spring lose stability quickly.

To propagate from a simulation state from $t_{n-1} \rightarrow t_n$ it is best to use some form of numerical integration (technically, we suppose, the last simulation was using integration...). Trick has a built in mechanism that supports several different types of integrators.

On this version of the cannonball, Trick's built-in integration scheme will be used. To use Trick's integration we will need to create:

1. A "derivative" class job for calculating acceleration.
2. An "integration" class job for integrating acceleration to velocity and velocity to position from time $t_{n-1}$ to $t_n$.
3. An input file, or default data file, for configuring the integration scheme.

## 5.1 Derivative Class Jobs - Finding Acceleration

Derivative class jobs are responsible for calculating acceleration. In the case of the cannonball, this is very simple. Acceleration is 0.0 in the X direction (assuming no wind resistance). Acceleration in the Y direction is -g. This yields a derivative class job with only two lines.

### 5.1.1 Spring Example

If we had a perfect spring, we would calculate acceleration as follows.

```
f = ma
f = k*x
--> a = k*x/m
```

Our derivative class job would have one line, maybe:

```
S->acceleration[0] = S->k*S->position[0]/S->mass ;
```

### 5.1.2 Space Shuttle Robotic Arm Example

Now we are talking! In a complex example like a robotic arm the derivative class job still boils down to simple Newtonian physics: acceleration = force/mass. Modeling mass is complex in itself. Calculating force might take 100,000 lines of code! The simulation may be multi-body where forces are being calculated for independent components. The forces acting on a robotic arm, to name a few, are forces due to gravity, payload, boom flex, atmospheric drag, etc. In simulating these forces, simplifications are made --- just like the cannonball! For example, to simulate a gravity gradient the body might be assumed to have a uniform mass distribution. The earth might be reduced to be a spherical ball. Newtonian physics might be used, and relativity ignored!

To calculate acceleration in a complex system such as this it is possible to have many derivative class jobs. Integration for various sub-systems may be done with different integration schemes per component. A second order Runge_Kutta scheme might be sufficient for orbital mechanics, but a more sophisticated integration scheme may be necessary for arm flex.

Enough said... back to our perfect little cannonball!

### 5.1.3 Creating A Cannonball Derivative Class Job

**Figure 28 Cannonball Derivative Job - cannon_deriv.c**

```
/********************************** TRICK HEADER *************************
PURPOSE:    ( Try Trick integration )
*************************************************************************/
#include "../include/cannon.h"

int cannon_deriv(CANNON* C)
{
      C->acc[0] = 0.0 ;
      C->acc[1] = -9.81 ;

      return(0);
}
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`
**UNIX Prompt>** `vi cannon_deriv.c <edit and save>`

## 5.2 Integration Class Jobs - From Acceleration To Position

In the second try at creating the cannonball simulation, the standard textbook method for calculating position based on constant acceleration ($p_1 = p_0 + v_0t + 1/2a_0t^2$) was used.

We took the liberty of modeling a spring with the same state propagation. Observing Figure 29, it is evident that this technique for simulating a simple spring has a definite problem. Energy is added into the system. The model of the spring uses a scheduled job to calculate the states as did the cannonball on the second attempt (using dt).

We could try some other technique for calculating $p_1$. The following technique will actually help the spring, but ultimately will fail.

$$\text{Let } a = (a_0 + a_1)/2$$
$$v = (v_0 + v_1)/2$$

$$v_1 = v_0 + a*t$$
$$p_1 = p_0 + vt + 1/2at^2$$

The last method is a little better but still adds energy into the system. We might get desperate and do something really kludgy like adding a damping factor!

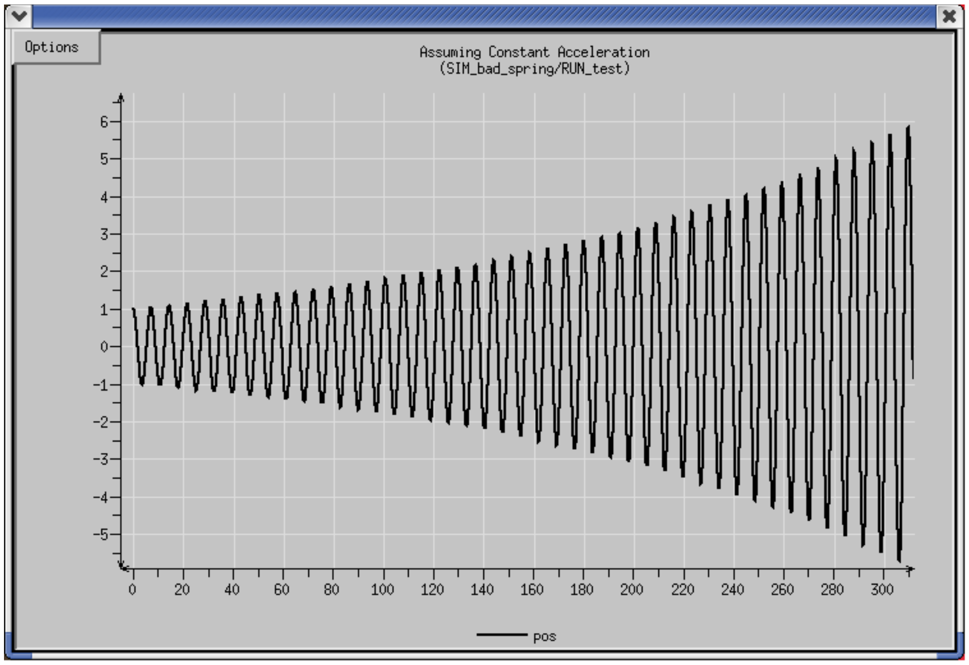
$$p_1 = (p_0 + v_0t + 1/2at^2)*(\text{damping factor})$$

Balking at something so kludgy, you might decide it best to break down and write an integrator! There are a couple of problems with creating your own integration scheme.

- Who wants to reinvent the wheel?
- The main problem is that putting your own integrator in a scheduled job doesn't work well for multiple pass integration schemes such as a Runge Kutta 4 algorithm.

- The other problem with having your own integrator in a scheduled class job is that integration should occur first off and should not be called at time=0.0. Remember in our second try of cannonball, the cannonball was well out of the barrel at time zero! We suppose you could argue that logging should occur before the job was run... and that would have fixed things. ;-)

**Figure 29 Spring Simulation Assuming Constant Acceleration**



## 5.2.1 Creating A Cannonball Integration Job

Figure 30 contains the code for the Trick integrator. What you probably notice right off is that we aren't in Kansas anymore. Here are a couple of observations about cannon_integ():

- This thing makes lots of references to a class called "INTEGRATOR" which we didn't create.
- It also calls a function called "integrate()" that we didn't create.
- It doesn't return zero, it returns "ipass".
- It isn't apparent what integration scheme is being used. Is it RK_2, RK_4...?

Let's unpack what is going on.

The first and second sections load the current states and derivatives. The states are simply loaded into a structure. This is equivalent to our scheduled job doing:

```
pos0[0] = C->pos[0] ; pos0[1] = C->pos[1] ;
vel0[0] = C->vel[0] ; vel0[1] = C->vel[1] ;
acc0[0] = C->acc[0] ; acc0[1] = C->acc[1] ;
```

The integrate() statement is equivalent to us calculating the current state based on the previous state. The integrate() routine is part of Trick core.

The section where the states are unloaded is necessary because the new states are contained in the structure. We need to copy them back out.

If this is a multi-pass integrator, cannon_integ() will call itself for each pass. The value returned from the integrate function will be non-zero until it reaches the number of passes required for the specified integration method. The Trick scheduler will stop calling cannon_integ() once it returns a value of zero.

The reason for loading the states into the Integrator class in the first place is simply because the integration is generic. The Integrator class is declared within the sim_services/Integrator/include directory. The integrate() routine knows how to manipulate data in the Integrator class.

**Figure 30 Cannonball Integration Job - cannon_integ.c**

```
/********************************** TRICK HEADER *************************
PURPOSE:    ( Try Trick integration )
*************************************************************************/
#include "sim_services/Integrator/include/integrator_c_intf.h"
#include "../include/cannon.h"
#include <stdio.h>

int cannon_integ(CANNON* C)
{
   int ipass;

   load_state(
      &C->pos[0] ,
      &C->pos[1] ,
      &C->vel[0] ,
      &C->vel[1] ,
      NULL
   );

   load_deriv(
      &C->vel[0] ,
      &C->vel[1] ,
      &C->acc[0] ,
      &C->acc[1] ,
      NULL
   );

   ipass = integrate();

   unload_state(
      &C->pos[0] ,
      &C->pos[1] ,
      &C->vel[0] ,
      &C->vel[1] ,
      NULL
   );

   return(ipass);

}
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`
**UNIX Prompt>** `vi cannon_integ.c <edit and save>`

### 5.2.2 Modify cannon_proto.h

**Figure 31 cannon_proto.h**

```
/*************************************************************
PURPOSE:      (Cannonball Prototypes)
*************************************************************/
#include "cannon.h"

#ifdef __cplusplus
extern "C" {
#endif

int cannon_integ(CANNON*) ;
int cannon_deriv(CANNON*) ;
int cannon_init(CANNON*) ;
int cannon_default_data(CANNON*);

#ifdef __cplusplus
}
#endif
```

## 5.3 A New Simulation Definition Which Includes Integration

In order to retain our old simulation tries, copy our last attempt.

**UNIX Prompt>** `cd $HOME/trick_sims`
**UNIX Prompt>** `cp -r SIM_cannon_dt SIM_cannon_integ`

The changes necessary to incorporate Trick integration into the S_define are highlighted in the following figure.

Once again add the source files to the header and the extern "C" block of code. In the CannonSimObject class add a pointer to the Trick::Integrator class and the pointers name can be any name of your choosing. Next, add the derivative and the integration jobs to the class. The derivative job only requires the derivative keyword in parenthesis, however, the integration job requires the integration keyword and the address of the Integrator pointer that was added above. Also, "trick_ret =" is required in the statement because Trick will use the return value to determine if the derivative/integration jobs need to be called again for the software frame. The statement at the bottom of the S_define "`IntegLoop dyn integloop (0.01) dyn;`" is outside the sim object. The keyword "`IntegLoop`" is mandatory. The next word is an instantiation of IntegLoop and may be any name of your choosing, however, if there are more than one IntegLoop statements the instantiation for each IntegLoop statement must be unique. In parenthesis is the frequency specification which tells Trick how often to integrate. Recall that neither cannon_deriv() or cannon_integ() made any reference to a time delta (dt). This is as it should be. The final keyword is the instantiation name of the sim object containing the integration jobs. Simulation objects are "integrated" as a whole.

The integration statement lumps together the derivative and integration class jobs. Derivative and integration are partners. They are the team for propagating the simulation state.

Note: By using Trick's integration, the scheduled task is no longer used to calculate the cannonball position as a function of time.

**Figure 32 Simulation Definition - S_define**

```
/*************************** Trick Header *********************************
PURPOSE: (S_define Header)
LIBRARY_DEPENDENCY: ((cannon/gravity/src/cannon_integ.c)
                     (cannon/gravity/src/cannon_deriv.c)
                     (cannon/gravity/src/cannon_init.c)
                     (cannon/gravity/src/cannon_default_data.c))
***************************************************************************
/
#include "sim_objects/default_trick_sys.sm"

##include "cannon/gravity/include/cannon.h"
##include "cannon/gravity/include/cannon_proto.h"

class CannonSimObject : public Trick::SimObject {
   public:
      CANNON cannon ;

      CannonSimObject() {
         ("initialization") cannon_init( &cannon ) ;
         ("default_data") cannon_default_data( &cannon ) ;
         ("derivative") cannon_deriv( &cannon ) ;
         ("integration") trick_ret = cannon_integ(&cannon) ;
      }
} ;

CannonSimObject dyn ;

IntegLoop dyn_integloop (0.01) dyn ;
```

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_integ`
**UNIX Prompt>** `vi S_define <edit and save>`

## 5.4 A New Input File Which Includes Integration

The input file needs to be updated to initialize the Integrator class. The following line needs to be added to the input file

**`dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 4)`**

Let's explain the above line a little further.

- `dyn_integloop – dyn_integloop` is the name of your IntegLoop from the S_define.
- `trick.Runge_Kutta_4 –` this is the integration method chosen for Trick to use.
- The second argument is the number of variables that are to be integrated. There are four variables for this simulation (pos[0], pos[1], vel[0], vel[1]).

  **UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_integ`
  **UNIX Prompt>** `vi RUN_test/input.py`

## 5.5 Running The Cannonball With Trick Integration

There is nothing different about running with Trick integration. We just need to build the simulation and run it.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_integ`
**UNIX Prompt>** `make spotless`
**UNIX Prompt>** `CP`
**UNIX Prompt>** `./S_main*exe RUN_test/input.py &`

Run the simulation to completion

### 5.5.1 Integration Versus Analytical

Let's compare the analytical "perfect" simulation with latest version using Trick integration.

Step 1. Start the trick data products

**UNIX Prompt>** `trick_dp &`

```
There should be the three SIMs in the "Sims/Runs" pane of trick_dp:
SIM_cannon_analytic, SIM_cannon_dt and SIM_cannon_integ.
```

Step 2. *Double click* `SIM_cannon_analytic->RUN_test`
This will move SIM_cannon_analytic/RUN_test to the selection box.

Step 3. *Double click* `SIM_cannon_integ->RUN_test`
Now the two RUNs we wish to compare will be present in the selection box.

Step 4. *Double click* `SIM_cannon_analytic/DP_cannon_xy`
`DP_cannon_xy` will be moved into the selection box.

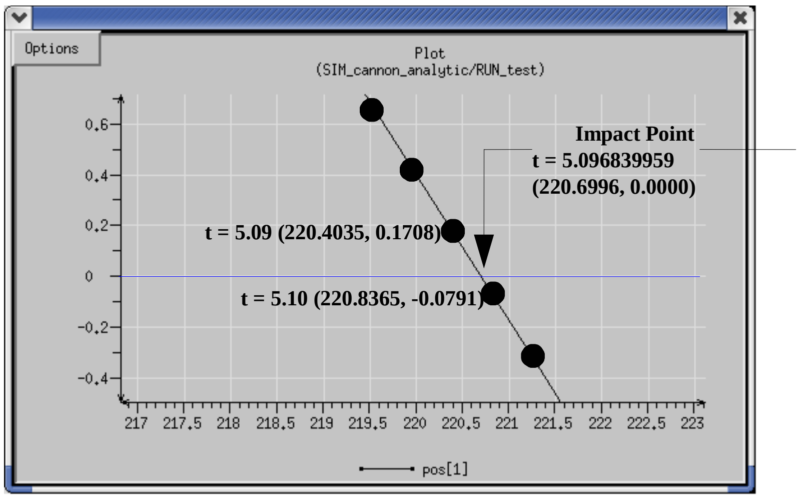
Step 5. *Click* the Co-Plot button (collated white sheets icon) located on the menu bar.
Voila! The curves appear the same, but there is a slight difference.

Living with less than a billionth of a meter difference will not cause us to lose sleep. However, we still don't like it! It is no fun being a sloppy perfectionist!

# 6.0 Contact - Using Dynamic Events

What is a cannonball that doesn't actually smack the earth and cause an immense explosion? Up until now, the simulation time was set to 5.2 seconds. An observant observer would have noticed that our cannonball has been penetrating the earth at about 5.1 seconds. If we were to run the simulation for 10 minutes or so, our cannonball would bury itself into the ground and burrow its way to China (assuming this tutorial isn't already on the other side of the world!). The Holy Hand Grenade of Antioch exploded on contact.

**Figure 33 Missed Contact**



Impact point

Time of impact (t) occurs when $p_y = 0.0$ or when

$v_{y0}t - 1/2gt^2 = 0$
$t(v_{y0} - 1/2gt) = 0$
$t = 2*v_{y0}/g$
t = 2*init_speed*sin(init_angle)/g
t = 2*50*cos(30)/g = 5.096839959 seconds

$p_{x\ impact} = v_{x0}$ * 5.096839959
= init_speed*sin(init_angle)*5.096839959
= 220.6996442

Since our integration frequency is 0.01, our simulation takes a discrete step from t=5.09 to t=5.10 and unknowingly passes ground level. What do we do? Here are some possibilities.

1. We could live with the fudge. We could assume contact occurs when the y position of the cannonball is <= 0.0. The penetration is only -0.0791 meters.

   Problem: Sometimes we can't live with the fudge! If we were simulating contact between a space station grapple mechanism and the pin on a satellite, would we really be comfortable with a fudge factor of -0.0791 meters or roughly 3 1/4 inches?!?

2. We could make our integration frequency higher. Instead of integrating at 0.01 seconds, we could integrate at 0.001 seconds or perhaps 0.00001 seconds!

   Problem: We went ahead and tried a few integration frequencies out.

   | | |
   |---|---|
   | 0.01 | >> -0.0791 |
   | 0.001 | >> -0.0040 |
   | 0.0001 | >> -0.0015 |
   | 0.00001 | >> -0.000001019 |

   This is dire. We need to have an integration frequency of 0.00001 seconds to get an answer that we might be able to live with. It would be ridiculous to have to integrate at this frequency just to get the contact to work out right. It is similar to buying a vehicle with a v12 with the thought -one- day you might actually need it! Hmmmm. That analogy might not hold :-)

3. We could look ahead one step each time. If we knew contact was going to happen on the following step,

we could refine our integration frequency to integrate to the point of contact.

Problem: This seems promising. The problem is that we would have to build some intelligence into our jobs for prediction and correction. This could get awful messy.

4. We could also pass through the point of contact, say to ourselves, "Woops" and try to fix the state of the simulation. In the cannonball situation we wouldn't accept the negative y position, we'd just set it to 0.0 if it ever got below the ground.

   Problem: This seems inviting. It sure is simple. It does force things to be correct, well sort-of correct. The problem is that we lose some of the important dynamics as the contact occurs. We have stripped some of the continuity of the dynamics. Our state is now a bit off. If we were simulating a ball bouncing along the ground, or a pin pinging its way down a V-guide, our simulation would slowly (or quickly) lose its integrity.

## 6.1 What Trick Offers For Contact - Dynamic Events

From the last section, door number 3 looked the most promising. Thankfully, Trick has a built in mechanism for iteratively calling its derivative and integration class job to accurately simulate a contact event. The mechanism is called "dynamic events". A dynamic event job will continually refine its own integration step-size based on a user defined tolerance. Dynamic events handle contact and other events that may not occur on an even integration state boundary. Similar to "integration" class jobs, the "dynamic event" jobs require an input file to configure the event. Configuration of the event includes a tolerance or epsilon of how accurate to carry out the integration refinement. Examples will follow!

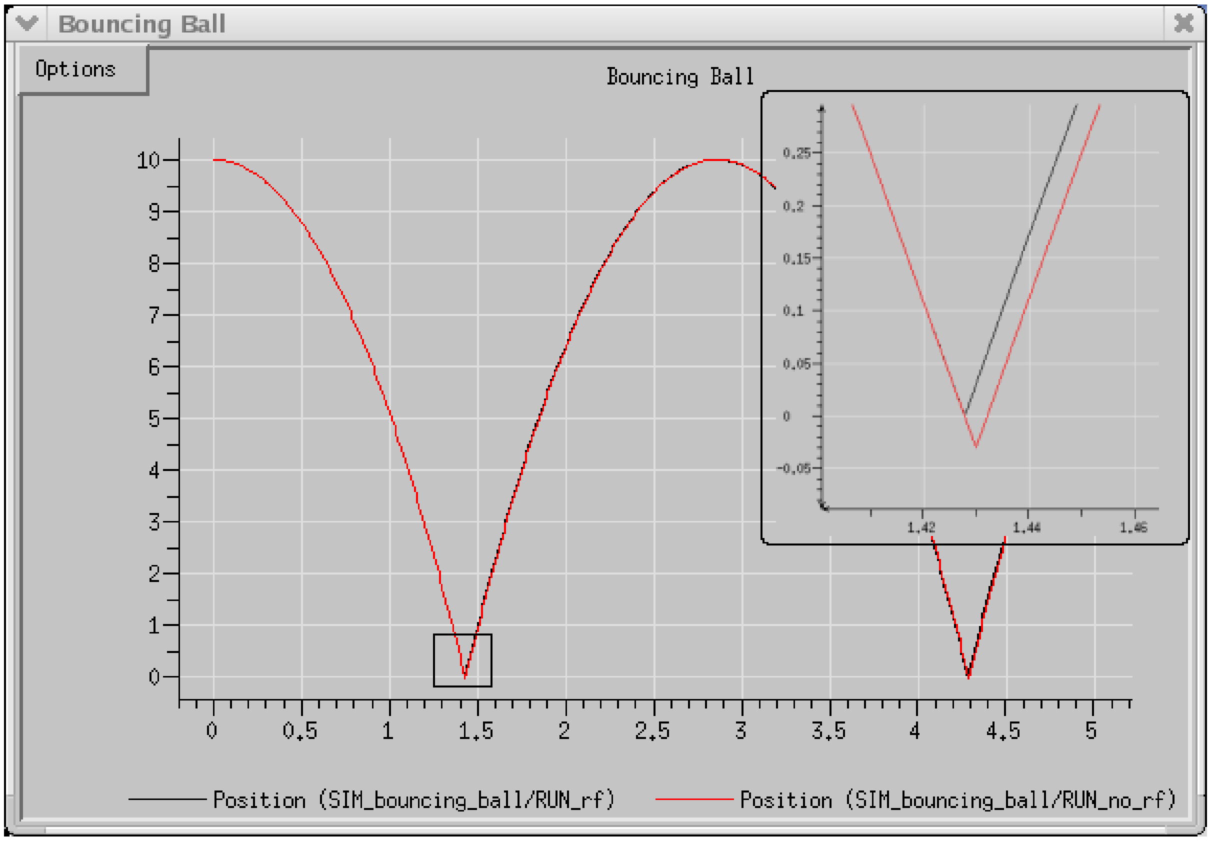
## 6.2 Bouncing Ball Example

To illustrate the issue of contact, we made two Trick simulations of a bouncing ball. The ball is dropped from 10 meters and bounces indefinitely with totally elastic collisions. In the first simulation, we used Trick integration and reversed velocity when the position of the ball was at or below floor level. The second simulation, we used Trick's dynamic events.

**Figure 34 Period Of Ball Bounce Calculated**

```
The ball is dropped from 10 meters.  What is its period, and where is at after 1000
bounces

Supposing initial velocity is 0.0,
the ball hits the floor when: 1/2gt^2 = 10.0, or t = sqrt(20/g)
It bounces the 2nd time at: 3*sqrt(20/g)
The nth bounce occurs at time: (2n-1)*sqrt(20/g)

The 1000th bounce occurs at about 2854.25840
```

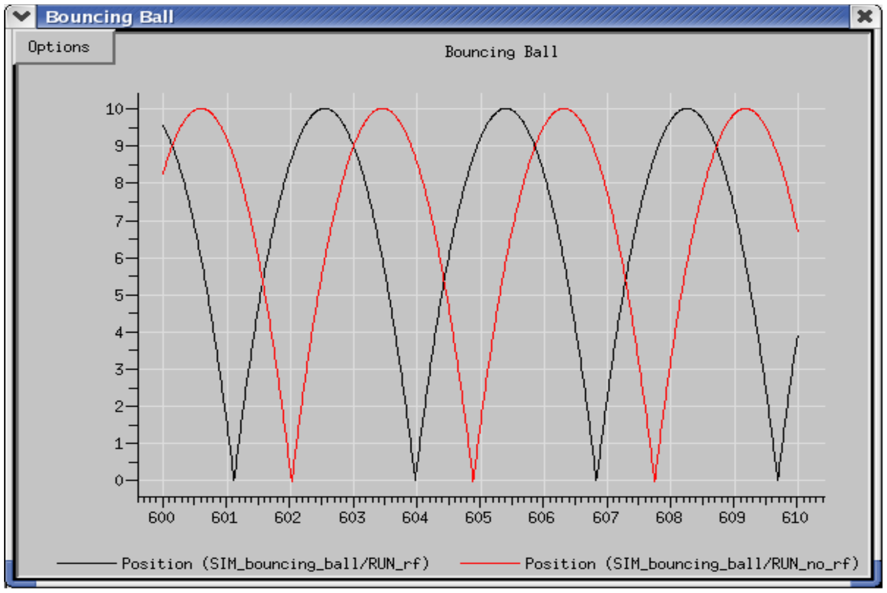
**Figure 35 Co-Plot Of First Two Bounces**



From a distance the comparison plot of the two methods looks fairly accurate. However, upon closer inspection we find a significant difference after a single bounce. The balls are already out-of-phase with one another. The phase difference is about .004 seconds.

The next plot shows the two simulations side-by-side after 5 minutes of execution. By this time the phase difference is about 1 second. Interestingly enough, after running this simulation for about 30,000 seconds or about 8 hours, the amplitude of the bounce never deviated. Both balls returned to 10 meters faithfully.

**Figure 36 Phase Difference After Running Five Minutes**



### 6.2.1 Bouncing Ball Accuracy Using Trick's Dynamic Events

We had calculated that the ball's 1000th bounce should occur at `2854.258403` seconds. Indeed, using Trick's dynamic events, the 1000th bounce occurs at `2854.258402731` seconds.

## 6.3 Implementing Contact In The Cannonball Simulation

### 6.3.1 Creating The Sim With Impact

This section will show the technicalities in setting up a "dynamic event" job. Specifically, we will setup a dynamic event job to handle the impact of the cannonball. We need to:

1. Make "dynamic event" job.
2. Update the cannon.h header to include impact variables (REGULA_FALSI structure).
3. Make an input file to configure the "dynamic event" structure.
4. Update the S_define to include the "dynamic event" job.

The next figure lists the contents of the "dynamic event" job that will handle impact. Here is a list of things to take note:

- The items in **bold face** are mandatory, and are unique to the "dynamic event" type job.
- The green block is where the error function is loaded and called. In the case of the cannonball, the distance between the cannonball and the ground is our error function. This distance is fed to the ever strange and mysterious "regula_falsi()" function. Like the integrate() function, the regula_falsi() function is buried in Trick. No need to write it!
- The red block is called once the error function value hits within an epsilon of zero. In the case of the cannonball, acceleration and velocity are set to zero. The C->impact flag is set to indicate that the impact has occurred. Two print() statements are called at the time of impact so you can see when the event fires. Later you will find that the time of impact is not on the 0.01 boundary, but is the exact value --- 5.096839959 seconds.

**Figure 37 Cannonball Dynamic Event Job - cannon_impact.c**

```
/*****************************************************************
PURPOSE:          ( Contact )
*****************************************************************/
#include <stdio.h>
#include "../include/cannon.h"

#include "sim_services/Integrator/include/integrator_c_intf.h"

double cannon_impact( CANNON* C )
{

   double tgo ;
   double now ;

   /* Set error function --- how far ball is above ground */
   C->rf.error = C->pos[1] ;

   now = get_integ_time() ;
   tgo = regula_falsi(now, &(C->rf) ) ;

   if (tgo == 0.0) {

      /* Cannonball hits dirt and goes BOOM. */
      C->impact = 1;

      now = get_integ_time() ;
      reset_regula_falsi( now, &(C->rf) )  ;

      C->vel[0] = 0.0 ; C->vel[1] = 0.0 ;
      C->acc[0] = 0.0 ; C->acc[1] = 0.0 ;

      fprintf(stderr,"\nImpact time = %.9lf\n", now) ;
      fprintf(stderr,"Impact pos = %.7lf\n",C->pos[0]);
   }
   return(tgo) ;
}
```

CALCULATE EPSILON

FIRE EVEN !!

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`

**UNIX Prompt>** `vi cannon_impact.c <edit and save>`

**Figure 38 Cannonball Impact Header Addition – cannon.h**

```
/********************************** TRICK HEADER *************************
PURPOSE:     (Cannonball Structure)
**********************************************************************/

#ifndef _cannon_h_
#define _cannon_h_

#include "sim_services/Integrator/include/regula_falsi.h"
#include “sim_services/include/Flag.h”

typedef struct {

      double pos0[2] ;   /* *i m Init position of cannonball */
      double vel0[2] ;   /* *i m/s Init velocity of cannonball */
      double acc0[2] ;   /* *i m/s2 Init acceleration of cannonball */

      double pos[2] ;   /*  m xy-position */
      double vel[2] ;   /*  m/s xy-velocity */
      double acc[2] ;   /*  m/s2 xy-acceleration  */
      double init_speed ; /* *i m/s Init barrel speed */
      double init_angle ; /* *i r Angle of cannon */

      /* Impact */
      REGULA_FALSI rf ; /* -- Dynamic event  params for impact */
      int impact ;     /* -- Has impact occured? */

} CANNON ;

#endif
```

Not too much to change to our CANNON structure. We just need to add in the enigmatic REGULA_FALSI structure. The REGULA_FALSI structure resides in Trick’s core package. You do NOT need to reset your TRICK_CFLAGS for Trick’s CP to find "sim_services/include/regula_falsi.h". Trick automatically searches in its core area.

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/include`
**UNIX Prompt>** `vi cannon.h <make modifications and save>`

**Figure 39 Cannonball Impact Default File – cannon_default_data.c**

```
/
       .
       .

int cannon_default_data(CANNON* C)
{
   const double PI = 3.141592 ;

   C->pos[0]        = 0.0 ;
   C->pos[1]        = 0.0 ;
   C->acc[0]        = 0.0 ;
   C->acc[1]        = -9.81 ;
   C->init_angle    = PI/6 ;
   C->init_speed    = 50.0 ;

    /* Regula Falsi dynamic event impact setup */
   #define BIG_TGO 10000
   C->rf.lower_set    = No ;
   C->rf.upper_set    = No ;
   C->rf.iterations   = 0 ;
   C->rf.fires        = 0 ;
   C->rf.x_lower      = BIG_TGO ;
   C->rf.t_lower      = BIG_TGO ;
   C->rf.x_upper      = BIG_TGO ;
   C->rf.t_upper      = BIG_TGO ;
   C->rf.delta_time   = BIG_TGO ;
   C->rf.error_tol = 1.0e-9 ;
   C->rf.mode      = Decreasing ;

   C->impact          = No ;

   return(0) ;
}
```

The items in bold are the two settings that warrant observation.

- error_tol
  This tells Trick how accurately to refine itself for firing the event. In this case, Trick will continually refine its integration until the error we send it is within 0.000000001. That is, when the cannonball's center reaches a height of 1 nanometer, impact occurs.

- mode
  Possible modes are "Decreasing", "Increasing" or "Any". In this case, the event should fire if the error is "decreasing" towards zero (i.e., the ball is dropping). We do not want to fire an event when we leave the cannon (the y-position is zero there too!). If we were going to bounce off of a ceiling we would want to put "Increasing" since our error function would be increasing in value when it hits. Specifying "Any" will cause the event to fire when either increasing or decreasing.

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`
**UNIX Prompt>** `vi cannon_default_data.c <modify and save>`

**Figure 40 Cannonball Impact Derivative Job - cannon_deriv_impact.c**

```
/*************************************************************
PURPOSE:         (Adding Contact)
*************************************************************/
#include "../include/cannon.h"

int cannon_deriv_impact( CANNON* C )
{
   if ( ! C->impact ) {
      /* Still above ground and flying */
      C->acc[0] = 0.0 ;
      C->acc[1] = -9.81 ;
   }
   return (0) ;
}
```

The derivative class job stays very simple. The derivative class job is responsible for calculating acceleration. If the impact has NOT occurred, it will remain the same as the impactless derivative class job. If the impact HAS occurred, acceleration will be set to zero by the "dynamic event" job... and it will remain zero. The flag C->impact is set in the "dynamic event" job.

The job has been renamed from "cannon_deriv()" to "cannon_deriv_impact()".

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`
**UNIX Prompt>** `vi cannon_deriv_impact.c <edit and save>`

**Figure 41 cannon_proto.h**

```
/*******************************************************
PURPOSE:      (Cannonball Prototypes)
*******************************************************/
#ifndef _cannon_proto_h_
#define _cannon_proto_h_

#include "cannon.h"

#ifdef __cplusplus
extern "C" {
#endif

int cannon_integ(CANNON*) ;
int cannon_deriv_impact(CANNON*) ;
double cannon_impact(CANNON*) ;
int cannon_init(CANNON*) ;
int cannon_default_data(CANNON*);

#ifdef __cplusplus
}
#endif
#endif
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/include`
**UNIX Prompt>** `vi cannon_proto.h <edit and save>`

**Figure 42 Simulation Definition For Impact - S_define**

```
/*************************** Trick Header *********************************
PURPOSE: (S_define Header)
LIBRARY_DEPENDENCY: ((cannon/gravity/src/cannon_integ.c)
                     (cannon/gravity/src/cannon_deriv_impact.c)
                     (cannon/gravity/src/cannon_impact.c)
                     (cannon/gravity/src/cannon_init.c)
                     (cannon/gravity/src/cannon_default_data.c))
**************************************************************************/
#include "sim_objects/default_trick_sys.sm"

##include "cannon/gravity/include/cannon.h"
##include "cannon/gravity/include/cannon_proto.h"

class CannonSimObject : public Trick::SimObject {
   public:
      CANNON cannon ;

      CannonSimObject() {
         ("initialization") cannon_init( &cannon ) ;

         ("default_data") cannon_default_data( &cannon ) ;

         ("derivative") cannon_deriv_impact( &cannon ) ;

         ("integration") trick_ret = cannon_integ(&cannon) ;

         ("dynamic_event") cannon_impact( &cannon ) ;
      }
} ;
CannonSimObject dyn ;

IntegLoop dyn_integloop (0.01) dyn ;
```

**UNIX Prompt>** `cd $HOME/trick_sims`
**UNIX Prompt>** `cp -r SIM_cannon_integ SIM_cannon_contact`
**UNIX Prompt>** `cd SIM_cannon_contact`
**UNIX Prompt>** `vi S_define <Make mods and save>`
**UNIX Prompt>** `make spotless`
**UNIX Prompt>** `CP`

## 6.3.2 Running The Sim With Impact

The simulation is ready to go! So FIRE!

**UNIX Prompt>** `S_main*exe RUN_test/input.py`

After running this simulation you should see a print out in the X-terminal where you launched the simulation. The printout reads:

```
IMPACT: time = 5.096838998, pos = 220.699616425
```

### 6.3.3 Viewing Data With Data Products

Great! Let's launch our data products and see how it looks.

Step 1. Launch Trick Data Product GUI

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_contact`

**UNIX Prompt>** `trick_dp &`

Step 2. *Choose* `SIM_cannon_contact->RUN_test`

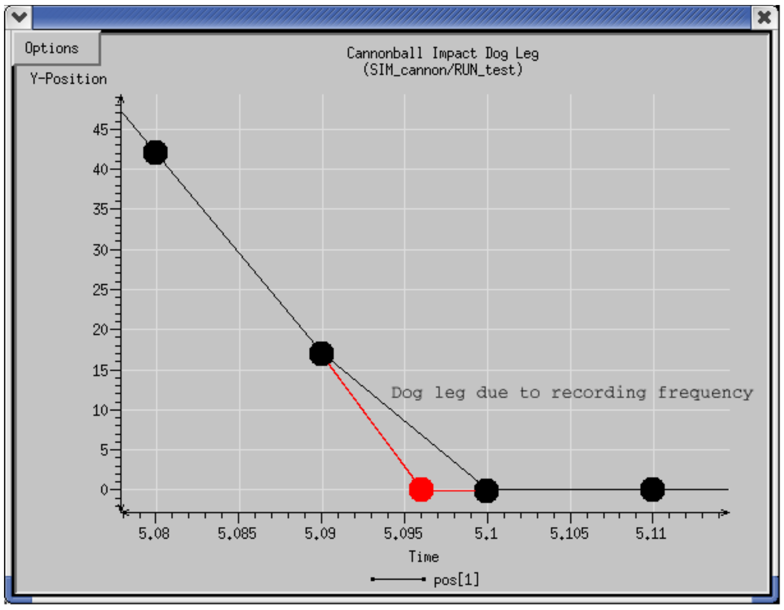
Step 3. Hit Quickplot button (blue lightning bolt)

Step 4. *Right click* `"dyn.cannon.pos[0-1]"` and *choose* expand var

Step 5. *Double click* `"dyn.cannon.pos[1]"`

Step 6. *Click* "Single" plot mode button

**Figure 43 Impact Plot- Dog Leg!**



The red shows what actually happened. Since we are recording at 0.01, the impact isn't recorded until 5.1 seconds whereas it occurred at 5.096839959 seconds.

Unfortunately there is no clean way to record what is going on during the impact calculations. However, the simulation's dynamics are accurate. The accuracy was illustrated in the example simulation where we bounced a ball 1000 times. In fact, the bouncing ball example DID record on time stamps that were not on 0.01 boundaries. We had to modify the job to record during collisions. It just wasn't very pretty, so we left it out of this example.

# 7.0 Communications

Up until this point the cannonball simulation has been an island unto itself. If we wanted to send the state of our cannonball to another program such as a graphics server we would need a job for communicating with the graphics server. If we wanted to drive our cannonball with a cannonball cockpit, go figure, we would need a way for the cockpit to send control commands to the cannonball.

Actually, the preceding paragraph is a little untrue. Our cannonball simulation has not been an island to itself, it has been communicating with an external program. The cannonball simulation has been sending position data to the stripchart program via Trick communications. The "sim control panel" is another example. The simulation control panel has been driving the simulation as well as receiving data via Trick communications.

There are a couple ways to communicate with the cannonball simulation:

1. Trick’s socket communications library.
2. We can make another process a "child" of the simulation which shares the same address space. This would be **direct** communication! A Trick "child" is actually a Posix thread.
3. Trick’s distributed simulation capability offers a convenient way for distributed simulations to pass data amongst themselves.
4. Write your own communication package! How about passing data on reflected memory boards?!? Or a 1553 bus??? Use your own socket library!

## 7.1 Simple Trick Server

**Figure 44 Trick Server Code - tc_server.c**

```
#include <stdio.h>
#include <string.h>
#include "trick_utils/comm/include/tc.h"
#include "trick_utils/comm/include/tc_proto.h"

int main (int narg, char** args)
{
   double x, y;
   int nbytes ;

   TCDevice listen_device ;
   TCDevice connection ;

   /* Zero out devices */
   memset(&listen_device, '\0', sizeof(TCDevice)) ;
   memset(&connection,    '\0', sizeof(TCDevice)) ;

   /* Accept connection */
   listen_device.port = 9000;
   tc_init(&listen_device) ;
   tc_accept(&listen_device, &connection) ;

   /* Read and display position data from client */
   while(tc_isValid(&connection)) {
      nbytes = tc_read(&connection, (char*) &x, sizeof(double));
      nbytes = tc_read(&connection, (char*) &y, sizeof(double));
      fprintf(stderr, "x=%lf y=%lf \n", x, y) ;
   }

   return 0 ;
}
```

This is a standalone main() program that accepts a client on port `9000`. "TCDevice" is the data structure that Trick uses for its connection management. Trick servers always "listen" on one "device", and perform all other communications on another "device". The listen device is used so that the server can accept multiple connections from multiple clients. All clients will connect to the server’s listen device. In this case, there is only one client, but nevertheless, a "listen device" is mandatory.

Once a connection is established through the tc_accept() call, the server will continually reads two double precision numbers from a client until the client breaks off the connection.

By default, the "connection" is blocking. The tc_read() calls will pause until the client sends its position data.

This example server is functional; however, it has no error checking. This was done for clarity. Beware!!!

To build this server, we need access to Trick's communication library and the Trick header tc.h. Before you can compile the stand-alone server, you will have to build the Trick's communication library as a stand-alone library.

**UNIX Prompt>** `cd $TRICK_HOME/trick_source/trick_utils/comm`
**UNIX Prompt>** `make STAND_ALONE=1`

Now that you have created the stand-alone communications library, you may compile tc_server.

```
UNIX Prompt>  cd $HOME
UNIX Prompt>  vi tc_server.c <edit and save>
UNIX Prompt>  cc -I$TRICK_HOME/trick_source -o tc_server tc_server.c \
      -L$TRICK_HOME/trick_source/trick_utils/comm/object_$TRICK_HOST_TYPE \
      -ltrick_comm –lrt
```

**Note**: `–lrt` is only needed for Linux users.

## 7.2 Simple Trick Client

**Figure 45 Trick Client Code - tc_client.c**

```
#include <string.h>
#include "trick_utils/comm/include/tc.h"
#include "trick_utils/comm/include/tc_proto.h"

int main( int narg, char** args) {
   TCDevice connection;
   double x, y ;

   /* Zero out device */
   memset(&connection, '\0', sizeof(TCDevice)) ;

   /* Set server address */
   connection.port = 9000;
   connection.hostname = (char*) malloc(16);
   strcpy(connection.hostname, "localhost");

   /* Connect to server */
   tc_connect(&connection);

   /* Round trip to server */
   x = 123.45;
   y = 67.89 ;
   tc_write(&connection, (char*) &x, sizeof(double));
   tc_write(&connection, (char*) &y, sizeof(double));

   /* Disconnect from server */
   tc_disconnect(&connection) ;

   return(0);
}
```

Short and sweet. Connect to the server on port 9000. Then tc_write() two double precision numbers.

```
UNIX Prompt>  cd $HOME
UNIX Prompt>  vi tc_client.c
UNIX Prompt>  cc -I$TRICK_HOME/trick_source -o tc_client tc_client.c \
       -L$TRICK_HOME/trick_source/trick_utils/comm/object_$TRICK_HOST_TYPE \
       -ltrick_comm –lrt
```

Note: `–lrt` is only needed for Linux users.

## 7.3 Running The Standalone Client And Server

To run the two main() programs follow the steps below. It is best to run the client and server in separate x-terminals.

Step 1. Bring up an x-terminal for server program

```
UNIX Prompt>  cd $HOME
UNIX Prompt>  ./tc_server
```

Step 2. Bring up another x-terminal for client program

```
UNIX Prompt>  cd $HOME
UNIX Prompt>  ./tc_client
```

The server should output:

```
ADVISORY: src/tc_accept.c:172: (ID = 0  tag = <empty>): connected to client

x=123.450000 y=67.890000
```

Ignore the advisory. It is the default for the server to print out an "advisory" when a client connects.

## 7.4 Connecting The Simulation To The Server

Now that we have a Trick client/server talking, let's hook up our cannonball simulation to the simple server. There are a couple more things to add to our cannonball simulation!

- Add a communication device to our CANNON structure for the client to use.
- Create a job that functions as a client to the position-printing server. This client will be able to pass current cannonball positions to the server.
- Launch the server in its own terminal so that the data displayed by the print() statement will be visible.

The standalone client written in section 7.2 will have to be broken into two parts. The first part will be connecting to the server. This only needs to be done once at simulation initialization, and therefore will be made into an initialization class job. The second part will be writing the position data as the simulation runs. Position data will be sent to the server via a "scheduled" class job.

## 7.4.1 Cannonball's Communication Device

**Figure 46 TCDevice For Cannonball - cannon.h**

```
/********************************** TRICK HEADER *************************
PURPOSE:     (Cannonball Structure)
*************************************************************************/
#ifndef _cannon_h_
#define _cannon_h_

#include <stdio.h>
#include "sim_services/Integrator/include/regula_falsi.h"
#include "sim_services/include/Flag.h"
#include "trick_utils/comm/include/tc.h"
#include "trick_utils/comm/include/tc_proto.h"

typedef struct {
      double pos0[2] ;   /* *i m Init position of cannonball */
      double vel0[2] ;   /* *i m/s Init velocity of cannonball */
      double acc0[2] ;   /* *i m/s2 Init acceleration of cannonball */

      double pos[2] ;   /*  m xy-position */
      double vel[2] ;   /*  m/s xy-velocity */
      double acc[2] ;   /*  m/s2 xy-acceleration  */
      double init_speed ; /* *i m/s Init barrel speed */
      double init_angle ; /* *i r Angle of cannon */

      /* Impact */
      REGULA_FALSI rf ; /* -- Dynamic event  params for impact */
      int impact ;     /* -- Has impact occured? */

      /* Communication Connection */
      TCDevice connection ; /* -- Socket connection for sending position */

} CANNON ;

#endif
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/include`
**UNIX Prompt>** `vi cannon.h <modify and save>`

**Figure 47 cannon_proto.h**

```
/*****************************************************************
PURPOSE:           (Cannonball Prototypes)
*****************************************************************/
#ifndef _cannon_proto_h_
#define _cannon_proto_h_
#include "cannon.h"

#ifdef __cplusplus
extern "C" {
#endif

int cannon_integ(CANNON*) ;
int cannon_deriv_impact(CANNON*) ;
double cannon_impact(CANNON*) ;
int cannon_init(CANNON*) ;
int cannon_default_data(CANNON*);
int cannon_init_comm(CANNON*) ;
int cannon_send_position(CANNON*) ;

#ifdef __cplusplus
}
#endif
#endif
```

A new edition is added to the CANNON structure: "TCDevice connection". The "connection" will be used to send position data to the server.

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/include`
**UNIX Prompt>** `vi cannon_proto.h <modify and save>`

### 7.4.2 Job Source Code For Data Connection

**Figure 48 Initialize Cannon Client/Server - cannon_init_comm.c**

```
/********************************** TRICK HEADER *************************
PURPOSE: (Initialize communications with a server)
*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "../include/cannon.h"
#include "sim_services/include/release.h"

int cannon_init_comm(CANNON* C)
{
      int ret, num_attempts ;

      /* Launch server */
      system("tc_server &");

      /* Initialize client's connection to server */
      C->connection.port = 9000 ;
      C->connection.hostname = (char*) malloc( 16 ) ;
      strcpy( C->connection.hostname, "localhost");

      /* Shutup status messages */
      tc_error( &C->connection, 0 ) ;

      /* Client connect to server : try for 5 seconds */
      num_attempts = 0 ;
      while ( 1 ) {
            ret = tc_connect( &C->connection ) ;
            if ( ret == TC_SUCCESS ) {
                  break ;
            } else {
                  num_attempts++ ;
                  if ( num_attempts == 500000 ) {
                        fprintf(stderr, "Couldn't connect to server.../n");
                        exit(-1);
                  }
                  RELEASE_1(); /* Pause a microsecond */
            }
      }
      return 0 ;
}
```

The upper green block launches the server created in section 7.1. The server is launched as a separate program through a system() call. If the servered created earlier is not in your path, you may need to copy the `tc_server` to the directory from where you run your sim.

The lower blue block connects the simulation client to the server. The while() loop allows the client to continually try connecting while the server gets up and going.

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`
**UNIX Prompt>** `vi cannon_init_comm.c <edit and save>`

**Figure 49 Client Code For Writing Position To Server - cannon_send_position.c**

```
/********************************** TRICK HEADER *************************
PURPOSE: (Send position to server)
*************************************************************************/
#include "../include/cannon.h"

int cannon_send_position( CANNON* C )
{
      tc_write( &C->connection, (char *) &C->pos[0], sizeof(double)) ;
      tc_write( &C->connection, (char *) &C->pos[1], sizeof(double)) ;

      return(0) ;
}
```

This job is so short. Isn't it nice? Shouldn't all jobs be so short! This job will be called at the frequency specified in the S_define.

**UNIX Prompt>** `cd $HOME/trick_models/cannon/gravity/src`
**UNIX Prompt>** `vi cannon_send_position.c <edit and save>`

## 7.4.3 Updating The Simulation Definition File - S_define

Now that the model code is in place, the simulation definition file needs to be updated. There are two jobs to add to the S_define.

1. cannon_init_comm()
   It launches the position print() server, "tc_server". It also connects the simulation client to the tc_server program.
2. cannon_send_position()
   This will function as a "scheduled" class job AND and an "initialization" class job. It will run before time=zero, at time=zero and will run every 0.01 seconds. Scheduled class jobs run -after- derivative/integration class jobs. The only exception is at time=0.0, where derivative/integration class jobs are NOT run.

**Figure 50 Communicating Simulation Definition - S_define**

```
/*************************** Trick Header *********************************
PURPOSE: (S_define Header)
LIBRARY_DEPENDENCY: ((cannon/gravity/src/cannon_integ.c)
                     (cannon/gravity/src/cannon_deriv_impact.c)
                     (cannon/gravity/src/cannon_impact.c)
                     (cannon/gravity/src/cannon_init.c)
                     (cannon/gravity/src/cannon_init_comm.c)
                     (cannon/gravity/src/cannon_send_position.c)
                     (cannon/gravity/src/cannon_default_data.c))
**************************************************************************
/
#include "sim_objects/default_trick_sys.sm"

##include "cannon/gravity/include/cannon.h"
##include "cannon/gravity/include/cannon_proto.h"

Class CannonSimObject : public Trick::SimObject {
   public:
      CANNON cannon ;

      CannonSimObject() {
         ("initialization") cannon_init( &cannon ) ;

         ("initialization") cannon_init_comm( &cannon ) ;

         ("initialization") cannon_send_position( &cannon ) ;

         ("default_data") cannon_default_data( &cannon ) ;

         ("derivative") cannon_deriv_impact( &cannon ) ;

         ("integration") trick_ret = cannon_integ(&cannon) ;

         ("dynamic_event") cannon_impact( &cannon ) ;

         (0.01, "scheduled") cannon_send_position( &cannon ) ;
      }
} ;

CannonSimObject dyn ;

IntegLoop dyn_integloop (0.01) dyn ;
```

**UNIX Prompt>** `cd $HOME/trick_sims`
**UNIX Prompt>** `cp -r SIM_cannon_contact SIM_cannon_trickcomm`
**UNIX Prompt>** `cd SIM_cannon_trickcomm`
**UNIX Prompt>** `vi S_define <modify and save>`

### 7.4.4 Build And Run

With new source code and a remodeled S_define in hand, the cannon may be shot once again.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_trickcomm`
**UNIX Prompt>** `make spotless`
**UNIX Prompt>** `CP`
**UNIX Prompt>** `./S_main*exe RUN_test/input.py &`

When the simulation runs, the tc_server will output its position output from the simulation to stdout.

# 8.0 Atmospheric Forces - Using Trick's Collect Mechanism

It has been assumed all along that the cannonball flies free with no drag or lift due to atmosphere. This assumption will no longer be made. Let's add atmospheric lift and drag!

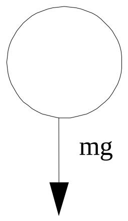
Besides the fun in simulating a curveball, the point of this section is to show how disparate forces can be collected into a derivative class job for acceleration calculations.

## 8.1 Throwing A Curveball - Shooting A Baseball

Are there any rules which prohibit shooting a baseball from a cannon? There is a lot more literature on the web about pitching a curve ball :-) There is also more experimental data to semi-verify the model. At least, we may find ourselves in the ballpark. Our reference might amaze you: *Alaways, L.W. 1998, Aerodynamics Of The Curve-Ball: An investigation of the effects of angular velocity on baseball trajectories. Unpublished doctoral dissertation, University Of California, Davis, CA.* We will take the liberty of referring to the cannonball as a baseball or as a ball.

The following diagrams show the forces on the baseball: $F_{total} = F_{gravity} + F_{drag} + F_{lift} + F_{cross}$. The forces are described in great detail in Alaway's paper. It is much too involved to include here. Never knew how complex such a "simple" problem could be.

**Figure 51 Gravitational Force**

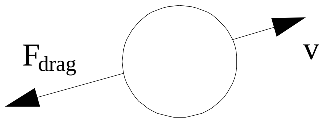


$$\vec{F}_{gravity} = mg$$

m = 0.145 kg
g = -9.81

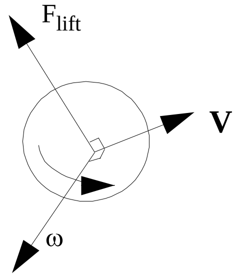
**Figure 52 Drag Force**

$$\vec{F}_{drag} = -\frac{1}{2}\rho C_D A V \vec{V}$$



We'll assume:
$\rho$ = Air density = 1.29kg/m$^3$ (density of air at 20ºC at sea level)
Radius of ball = 2.9/2.0 inches (baseball) = 2.9*2.54/2.0 = 3.683 cm
A = Cross sectional area of ball = 41.59cm$^2$
$C_D$ = Drag Coefficient = I read this is between 0.07 and 0.5.
Referring to Alaways, $C_D$ is roughly 0.45. It is very confusing though! There are dependencies on Reynold's number, viscosity of air, etc.

**Figure 53 Magnus/Lift Force- Default Method**

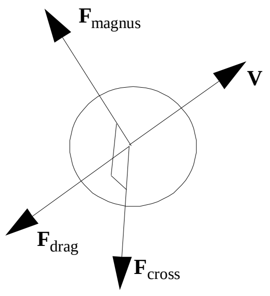


$$\vec{F}_{lift} = \frac{\rho C_L A V^2}{2} \frac{\vec{\omega} \times \vec{V}}{\left|\vec{\omega} \times \vec{V}\right|}$$

$$C_L = 0.54 S^{0.4}$$

$$S = \frac{r\omega}{V}$$

**Figure 54 Cross Force (Side Force)**



$$\vec{F}_{cross} = \frac{1}{2}\rho C_Y A V^2 \frac{\overrightarrow{F_{magnus}} \times \overrightarrow{F_{drag}}}{\left|\overrightarrow{F_{magnus}} \times \overrightarrow{F_{drag}}\right|}$$

$$C_Y = 0.044$$

$\rho$ = Air density = 1.29 kg/m3
$C_Y$ = Cross-force coefficient as seen in Alaway's paper
A = Cross sectional area of ball
V = Velocity of ball

## 8.2 Source Code

The following sections lay out all source code changes necessary for the new model.

To create a directory tree to hold the source, follow the instructions below:

```
UNIX Prompt>  mkdir -p $HOME/trick_sims/SIM_cannon_aero/RUN_curveball
UNIX Prompt>  mkdir -p $HOME/trick_sims/SIM_cannon_aero/Modified_data
UNIX Prompt>  mkdir -p $HOME/trick_models/cannon/aero/src
UNIX Prompt>  mkdir -p $HOME/trick_models/cannon/aero/include
```

### 8.2.1 Header Changes

In order to model atmospheric forces we need to extend/complicate the beloved simple two-dimensional cannonball we've grown so fond of. Here is a list of things to add:

1. Add another dimension. The cannonball was confined to XY. We need Z. Hard to make a curve ball in

2D.

2. Spin rate (as well as the orientation of spin axis) must be accounted for.
3. Environment and other properties now come into play. Example, the mass and radius of the cannonball had no bearing on the non-atmospheric model. Mass and shape now play a big role. We also have to concern ourselves with air density. We will also assume certain coefficients for drag and cross forces.
4. Keep track of individual forces (drag, lift, cross, and gravitational).

The items in the list above need to be incorporated into our new CANNON structure. To keep from confusing the old 2D CANNON structure with the new one which includes atmospheric drag, CANNON was renamed to CANNON_AERO. CANNON_AERO will also be put in a new header file called "cannon_aero.h".

**Figure 55 cannon_aero.h**

```
/*********************************** TRICK HEADER *************************
PURPOSE:                  (Test Baseball)
*************************************************************************/
#ifndef _cannon_aero_h_
#define _cannon_aero_h_

#include "sim_services/Integrator/include/regula_falsi.h"
#include "sim_services/include/Flag.h"

typedef enum {
      Hard_Coded_Coefficient_Lift,   /* You come up with Cl  */
      Smits_Smith,                   /* Cl ~= 0.54*S^0.4 (S = rw/V) */
      Adair_Giordano,               /* Lift_Force = mass*0.00041*w_cross_V */
      Tombras                       /* Cl = 1/(2.022 + 0.981v/w) */
} Lift_Estimation_Method ;

typedef struct {
      double pos[3] ;                /* m position */
      double vel[3] ;                /* m/s velocity */
      double acc[3] ;                /* m/s2 acceleration */
      double omega[3] ;              /* r/s Angular velocity of cannonball */

      double theta ;                 /* r Angle from x-axis to axis rotation */
      double phi ;                   /* r Angle from z-axis to axis rotation */
      double omega0 ;                /* r/s Magnitude of angular velocity about
                                              axis of rotation */
      /* Impact */
      REGULA_FALSI rf ;              /* -- Dynamic event params for impact */
      int impact ;                   /* -- Has impact occured */
      double impact_pos ;            /* m How far ball lands in field */

      /*  Forces */
      double force_gravity[3] ;      /* N Gravitational force */
      double force_drag[3] ;         /* N Drag force opposite dir of velocity */
      double force_magnus[3] ;       /* N Force due to spin, dir grav cross drag */
      double force_cross[3] ;        /* N Side directional force */
      double force_total[3] ;        /* N Sum of all forces */

      /* Magnitude */
      double mag_force_drag ;        /* N Magnitude of drag force */
      double mag_force_magnus ;      /* N Magnitude of magnus force */
      double mag_force_cross ;       /* N Magnitude of cross force */
      double mag_omega ;             /* r/s Magnitude of angular velocity */

      void ** force_collect ;        /* ** For collect statement */

      /* Environment and Properties */
      double mass ;                  /* kg Mass of cannonball */
      double air_density ;           /* kg/m3 Air density at 20C */
      double ball_radius ;           /* m Radius of cannonball */
      double ball_area ;             /* m2 Cross sectional area of ball */
      double spin_parameter ;        /* -- S=r*omega/speed */
      double g ;                     /* m/s2 Gravitational acceleration */

      /* Coefficients drag, lift and cross */
      Lift_Estimation_Method  lift_method ; /* -- How to find lift force */

      double coefficient_drag ;           /* -- Drag coefficient */
      double coefficient_lift ;           /* -- Lift coefficient */
      double coefficient_cross ;          /* -- Cross-Force coefficient */

} CANNON_AERO ;
#endif
```

#### 8.2.1.1 Collect Variable - void** force_collect ???

The "void** force_collect;" line was highlighted in the new version of cannon_aero.h. What could this possibly be? What is it for? Why void**? What on earth???

1. void** is a way of saying, "I want a generic pointer to a generic pointer. Later I'll decide what specific type I am pointing to."
2. There is nothing magical about the name "force_collect". We could have called it "mickey_mouse".
3. Its use will come to light later in the tutorial.

#### 8.2.1.2 Enumerated Types - Lift_Estimation_Method

There are several ways to estimate the lift force. In our model, we will get to choose between differing estimation techniques. This will also give us a chance to compare lift force calculation methods.

Lift_Estimation_Method was defined as:

```
typedef enum {
        Hard_Coded_Coefficient_Lift,        /* You come up with Cl    */
        Smits_Smith,                        /* Cl ~= 0.54*S^0.4 (S = rw/V) */
        Adair_Giordano,                     /* Lift_Force = mass*0.00041*w_cross_V */
        Tombras                             /* Cl = 1/(2.022 + 0.981v/w) */
} Lift_Estimation_Method ;
```

This isn't a typedef of a *struct*. This is a typedef of an *enum*. Yes! Trick understands enumerations. Later you will see how to use enumerations in the simulation input file.

#### 8.2.1.3 Edit cannon_aero.h

```
UNIX Prompt>  cd $HOME/trick_models/cannon/aero/include
UNIX Prompt>  vi cannon_aero.h <edit and save>
```

### 8.2.1.4 Edit cannon_aero_proto.h

You will be creating the jobs in cannon_aero_proto.h in next sub-section.

**Figure 56 cannon_aero_proto.h**

```
/*************************************************************
PURPOSE:              (Cannon_aero Prototypes)
*************************************************************/
#ifndef _cannon_aero_proto_h_
#define _cannon_aero_proto_h__
#include "cannon_aero.h"

#ifdef __cplusplus
extern "C" {
#endif

int cannon_init_aero(CANNON_AERO*) ;
int cannon_force_gravity(CANNON_AERO*) ;
int cannon_force_drag(CANNON_AERO*) ;
int cannon_force_lift(CANNON_AERO *);
int cannon_force_cross(CANNON_AERO *) ;
int cannon_collect_forces(CANNON_AERO *) ;
int cannon_integ_aero(CANNON_AERO*) ;
double cannon_impact_aero(CANNON_AERO*) ;
int cannon_aero_default_data(CANNON_AERO*) ;

#ifdef __cplusplus
}
#endif
#endif
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/include`

**UNIX Prompt>** `vi cannon_aero_proto.h <edit and save>`

## 8.2.2 Calculating Forces

The following sections list the source code for calculating forces and give instructions on where to place the new code.

### 8.2.2.1 Source Code For Calculating Forces

**Figure 57 Force Gravity - cannon_force_gravity.c**

```
/***********************************************************
PURPOSE:     ( Gravitational force on cannonball )
***********************************************************/
#include "../include/cannon_aero.h"

int cannon_force_gravity(CANNON_AERO *C)
{
        C->force_gravity[0] = 0.0 ;
        C->force_gravity[1] = 0.0 ;
        C->force_gravity[2] = C->mass*C->g ;

        return(0) ;
}
```

**Figure 58- Force Drag - cannon_force_drag.c**

```
/*********************************************************
PURPOSE:          (Drag Force)
*********************************************************/
#include "../include/cannon_aero.h"
#include "trick_utils/math/include/trick_math.h"

int cannon_force_drag(CANNON_AERO *C)
{
       double k ;
       double speed ;

       speed = V_MAG( C->vel ) ;

       /* k = -1/2*rho*Cd*A*|V| */
       k = (-0.5)*C->air_density*C->coefficient_drag*C->ball_area*speed ;

       /* Force_drag = k*V = -1/2*rho*Cd*A*|V|*V */
       V_SCALE( C->force_drag, C->vel, k ) ;

       C->mag_force_drag = V_MAG( C->force_drag ) ;

       return(0);
}
```

-

**Figure 59 Force Cross - cannon_force_cross.c**

```
/*******************************************************
PURPOSE:         (Cross Force or Side Force )
*******************************************************/
#include "../include/cannon_aero.h"
#include "trick_utils/math/include/trick_math.h"

int cannon_force_cross(CANNON_AERO *C)
{
       double magnus_cross_drag[3] ;
       double norm_magnus_cross_drag[3] ;
       double k, speed ;

       /* k = 1/2*rho*Cy*A*V^2 */
       speed = V_MAG( C->vel ) ;
       k = (-0.5)*C->air_density*C->coefficient_cross*C->ball_area*speed*speed ;

       /* F = k*(M x D)/|M x D| */
       V_CROSS( magnus_cross_drag, C->force_magnus, C->force_drag ) ;
       V_NORM( norm_magnus_cross_drag, magnus_cross_drag ) ;
       V_SCALE( C->force_cross, norm_magnus_cross_drag, k) ;

       C->mag_force_cross = V_MAG( C->force_cross ) ;

       return(0) ;
}
```

**Figure 60 Force Magnus/Lift - cannon_force_lift.c**

```
/*********************************************************
PURPOSE:         (Lift_Magnus Force)
*********************************************************/
#include "../include/cannon_aero.h"
#include "trick_utils/math/include/trick_math.h"

int cannon_force_lift(CANNON_AERO *C )
{
        double w_cross_v[3] ; double norm_w_cross_v[3] ;
        double k, speed ;

        speed = V_MAG( C->vel ) ;

        if ( speed != 0.0 ) {
              C->spin_parameter = C->ball_radius*V_MAG( C->omega )/speed ;
        } else {
              C->spin_parameter = 0.0000000001 ;
        }

        if ( C->lift_method == Smits_Smith ) {
              C->coefficient_lift = (0.54)*pow(C->spin_parameter, 0.4) ;
        }

        if ( C->lift_method == Tombras ) {
              C->coefficient_lift = 1/( 2.022 + 0.981*speed/V_MAG( C->omega));
        }

        switch ( C->lift_method ) {

           case  Hard_Coded_Coefficient_Lift:
           case  Smits_Smith:
           case  Tombras:
              /* k = 1/2*rho*Cl*A*V^2 */
              k = (0.5)*C->air_density*C->coefficient_lift*
                   C->ball_area*speed*speed ;

              /* F = k*(w x V)/|w x V| */
              V_CROSS( w_cross_v, C->omega, C->vel ) ;
              V_NORM( norm_w_cross_v, w_cross_v ) ;
              V_SCALE( C->force_magnus, norm_w_cross_v, k) ;
           break ;

           case Adair_Giordano:
              V_CROSS( w_cross_v, C->omega, C->vel ) ;
              k = 0.00041*C->mass ;
              V_SCALE( C->force_magnus, w_cross_v, k) ;

              /* Backwards calculation for Cl */
              C->coefficient_lift = (2*k*V_MAG(w_cross_v))/
                                    (C->air_density*C->ball_area*speed*speed) ;
           break ;
        }
        C->mag_force_magnus = V_MAG( C->force_magnus ) ;

        return 0 ;
}
```

#### 8.2.2.2 Editing The Force Source Code

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/src`
**UNIX Prompt>** `vi cannon_force_gravity.c (see Figure 57)`
**UNIX Prompt>** `vi cannon_force_drag.c (see Figure 58)`
**UNIX Prompt>** `vi cannon_force_cross.c (see Figure 59)`
**UNIX Prompt>** `vi cannon_force_lift.c (see Figure 60)`

### 8.2.3 Impact - Hitting The Catcher’s Mitt

The dynamic event job hardly changes at all. The only real change is due to an extra dimension. Gravitational pull is zeroed once it stops. This just keeps it from hitting the catcher’s mitt and falling on the ground. We’ll assume the guy can catch!

**Figure 61 Impact - cannon_impact_aero.c**

```
/*******************************************************
PURPOSE:                (Kaboom!!!)
*******************************************************/
#include <stdio.h>
#include “../include/cannon_aero.h”
#include "sim_services/Integrator/include/integrator_c_intf.h"

double cannon_impact_aero(  CANNON_AERO* C )
{
   double tgo ;
   double now ;

   /* Calculate time to hit catcher’s mitt (x pos == 0) */
   C->rf.error = C->pos[0] ;

   tgo = regula_falsi( get_integ_time() , &(C->rf) ) ;

   if (tgo == 0.0) {
      /*  Ball hits catcher’s mitt */
      now = get_integ_time() ;
      reset_regula_falsi( now , &(C->rf) ) ;

      C->vel[0] = 0.0 ; C->vel[1] = 0.0 ; C->vel[2] = 0.0 ;
      C->acc[0] = 0.0 ; C->acc[1] = 0.0 ; C->acc[2] = 0.0 ;
      C->g = 0.0 ;

      fprintf(stderr, "Impact time : %.9lf \n", now );
   }
   return(tgo) ;
}
```

#### 8.2.3.1 Edit cannon_impact_aero.c

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/src`
**UNIX Prompt>** `vi cannon_impact_aero.c <edit and save>`

### 8.2.4 Integration - Adding One More Dimension

The integration job almost stays the same. We just need to add the Z-dimension.

**Figure 62 Integration - cannon_integ_aero.c**

```
/*********************************************************
PURPOSE:            (Cannon Integration)
*********************************************************/
#include <stdio.h>
#include "sim_services/Integrator/include/integrator_c_intf.h"
#include "../include/cannon_aero.h"

int cannon_integ_aero(CANNON_AERO *C)
{
    int ipass;

    /* LOAD THE POSITION AND VELOCITY STATES */
    load_state(
        &C->pos[0] ,
        &C->pos[1] ,
        &C->pos[2] ,
        &C->vel[0] ,
        &C->vel[1] ,
        &C->vel[2] ,
        NULL
    );

    /* LOAD THE POSITION AND VELOCITY STATE DERIVATIVES */
    load_deriv(
        &C->vel[0] ,
        &C->vel[1] ,
        &C->vel[2] ,
        &C->acc[0] ,
        &C->acc[1] ,
        &C->acc[2] ,
        NULL
    );

    /* CALL THE TRICK INTEGRATION SERVICE */
    ipass = integrate();

    /* UNLOAD THE NEW POSITION AND VELOCITY STATES */
    unload_state(
        &C->pos[0] ,
        &C->pos[1] ,
        &C->pos[2] ,
        &C->vel[0] ,
        &C->vel[1] ,
        &C->vel[2] ,
        NULL
    );

    return(ipass) ;
}
```

### 8.2.4.1 Edit cannon_integ_aero.c

```
UNIX Prompt>  cd $HOME/trick_models/cannon/aero/src
UNIX Prompt>  vi cannon_integ_aero.c <edit and save>
```

## 8.2.5 Initialization Routine

Note that the CANNON_AERO structure defines the orientation and magnitude of the spin vector **omega** in terms of $\theta$, $\phi$, $\omega_o$ (magnitude of spin). The initialization routine will convert ($\theta$, $\phi$, $\omega_o$)–>($\omega_x$, $\omega_y$, $\omega_z$). It is basically a spherical coordinate to rectangular coordinate transformation.

**Figure 63 Ball Spin Vector - omega**

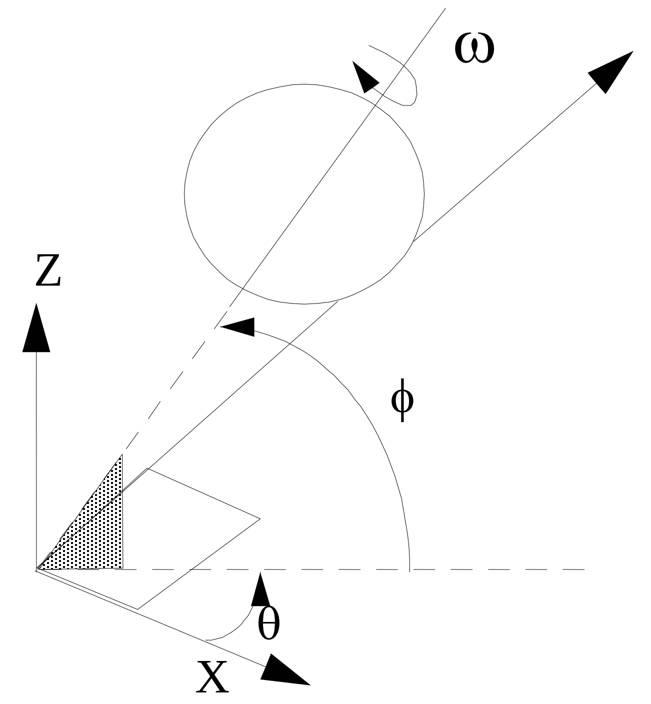


**Figure 64 Initialization Source - cannon_init_aero.c**

```
/*********************************************************
PURPOSE:      (Cannon Initialization)
*********************************************************/
#include <math.h>
#include <stdio.h>

#include "../include/cannon_aero.h"
#include "trick_utils/math/include/trick_math.h"

int cannon_init_aero(CANNON_AERO* C)
{
      /* Convert omega from spherical (almost) to rectangular */
      C->omega[0] = C->omega0*sin(M_PI/2.0 - C->phi)*cos(C->theta) ;
      C->omega[1] = C->omega0*sin(M_PI/2.0 - C->phi)*sin(C->theta) ;
      C->omega[2] = C->omega0*cos(M_PI/2.0 - C->phi) ;

      return(0) ;
}
```

### 8.2.5.1 Edit The Initialization Source Routine

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/src`
**UNIX Prompt>** `vi cannon_init_aero.c <edit and save>`

### 8.2.6 Default Data

We need to give the simulation some default values. Remember that the default values may be overwritten in the input file. This file sets up the base defaults. The default method for calculation lift force is *Smits_Smith*. The coefficients of drag and cross force are hard coded.

**Figure 65 Default Data Source – cannon_aero_default_data.c**

```
/**********************************************************
PURPOSE:            (Set the default data values)
**********************************************************/
#include "../include/cannon_aero.h"

int cannon_aero_default_data(CANNON_AERO* C)
{
   double const newton = 4.44822162 ;

   /* Initialize cannon ball shot */
   C->lift_method = Smits_Smith ;
   C->coefficient_drag = 0.45 ;
   C->coefficient_cross = 0.044 ;
   C->mass = 0.145 ;
   C->air_density = 1.29 ;
   C->ball_radius = 3.63/100 ;
   C->ball_area =  41.59/10000 ;
   C->g = -9.81;

   /* Regula Falsi impact critter setup */
   #define BIG_TGO 10000
   C->rf.lower_set = No ;
   C->rf.upper_set = No ;
   C->rf.iterations = 0 ;
   C->rf.fires = 0 ;
   C->rf.x_lower = BIG_TGO ;
   C->rf.t_lower = BIG_TGO ;
   C->rf.x_upper = BIG_TGO ;
   C->rf.t_upper = BIG_TGO ;
   C->rf.delta_time = BIG_TGO ;
   C->rf.error_tol = 1.0e-9 ;
   C->rf.mode = Decreasing ;

   return(0) ;
}
```

#### 8.2.6.1 Edit cannon_aero_default_data.c

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/src`

**UNIX Prompt>** `vi cannon_aero_default_data.c <edit and save>`

### 8.2.7 State Initialization

To initialize the state of the simulation, the ball needs an initial position, velocity, spin orientation, and spin velocity. A great place to put this information is in the simulation input file. This way the different pitches may be setup without having to recompile. To begin, the particular case shown below will be assumed.

**Figure 66 Initial State**

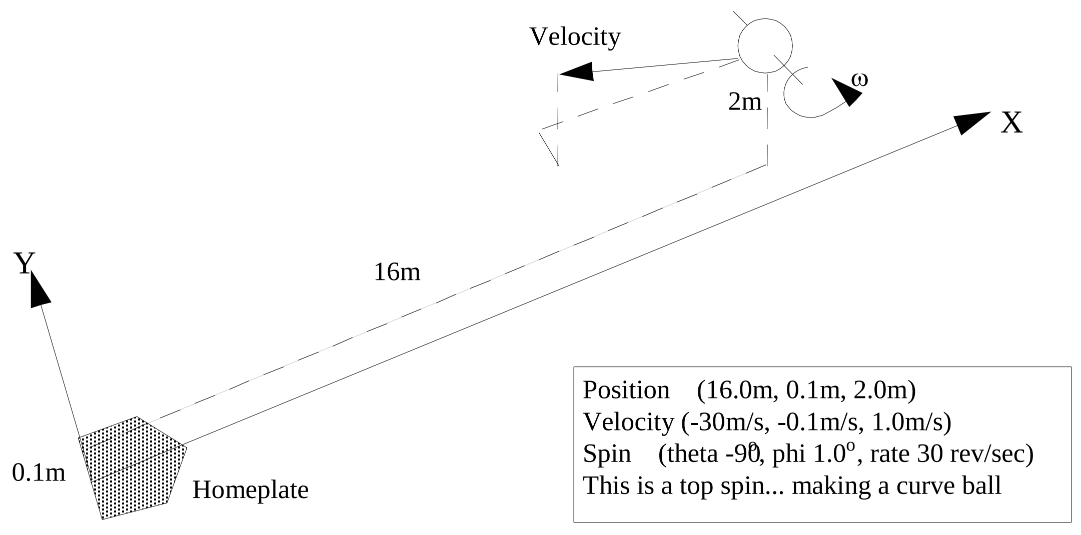

Velocity
ω
2m
X
Y
16m
0.1m
Homeplate
Position (16.0m, 0.1m, 2.0m)
Velocity (-30m/s, -0.1m/s, 1.0m/s)
Spin (theta -90°, phi 1.0°, rate 30 rev/sec)
This is a top spin... making a curve ball

**Figure 67 Initialization File – input.py**

```
dyn.baseball.pos[0] = 16.0
dyn.baseball.pos[1] = 0.1
dyn.baseball.pos[2] = 2.0

dyn.baseball.vel[0] = -30.0
dyn.baseball.vel[1] = -0.1
dyn.baseball.vel[2] = 1.0

dyn.baseball.theta = trick.attach_units("d",-90.0)
dyn.baseball.phi = trick.attach_units("d",1.0)
dyn.baseball.omega0 = trick.attach_units("rev/s",30.0)

dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 6)
trick.stop(5.2)
```

#### 8.2.7.1 Edit RUN_curveball/input.py

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_aero/RUN_curveball`

**UNIX Prompt>** `vi input.py <edit and save>`

### 8.2.8 Preliminary Simulation Definition

This is a bit premature, but let's go ahead and roll our source routines (jobs) and default data into a preliminary S_define. It is important to note that `cannon_force_cross()` follows `cannon_force_drag()` and `cannon_force_magnus()`. This is because cross force is dependent on drag and lift/magnus. Jobs of the same job class are executed in the order they are found in the S_define.

**Figure 68 Preliminary Simulation Definition - S_define**

```
/**************************** Trick Header ********************************
PURPOSE: (S_define Header)
LIBRARY_DEPENDENCY: ((cannon/aero/src/cannon_init_aero.c)
                     (cannon/aero/src/cannon_force_gravity.c)
                     (cannon/aero/src/cannon_force_drag.c)
                     (cannon/aero/src/cannon_force_lift.c)
                     (cannon/aero/src/cannon_force_cross.c)
                     (cannon/aero/src/cannon_integ_aero.c)
                     (cannon/aero/src/cannon_impact_aero.c)
                     (cannon/aero/src/cannon_aero_default_data.c)
                    )
***************************************************************************/
#include "sim_objects/default_trick_sys.sm"

##include "cannon/aero/include/cannon_aero.h"
##include "cannon/aero/include/cannon_aero_proto.h"

class CannonSimObject : public Trick::SimObject {
public:
   CANNON_AERO baseball ;

   CannonSimObject() {

      ("initialization") cannon_init_aero( &baseball ) ;             INIT
      ("default_data") cannon_aero_default_data( &baseball ) ;

      ("derivative") cannon_force_gravity( &baseball ) ;               FORCES
      ("derivative") cannon_force_drag( &baseball ) ;
      ("derivative") cannon_force_lift( &baseball ) ;
      ("derivative") cannon_force_cross( &baseball ) ;

      ("integration") trick_ret = cannon_integ_aero(&baseball) ;      INTEGRATION

      ("dynamic_event") cannon_impact_aero( &baseball ) ;             CONTACT
   }
} ;

CannonSimObject dyn ;

IntegLoop dyn_integloop (0.01) dyn ;
```

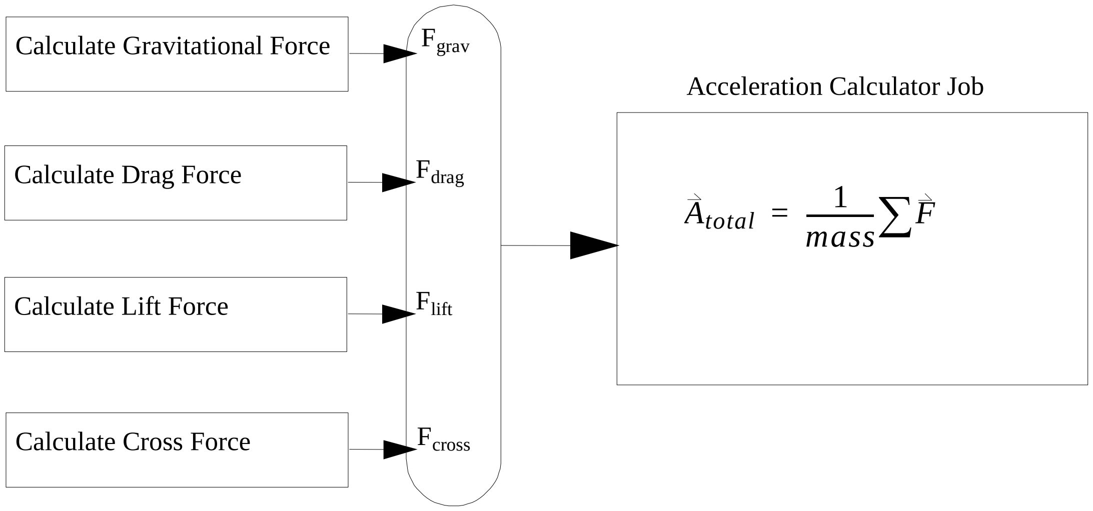
**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_aero`
**UNIX Prompt>** `vi S_define`

## 8.2.9 Calculating Acceleration

There is a missing piece to the puzzle. In the preliminary simulation definition file, forces were calculated using four derivative class jobs (drag, gravity, cross, lift). Then an integration job was called. The integration job is responsible for integrating current position, velocity and acceleration to a new position-velocity state. What about acceleration?! Isn't the purpose of our derivative class jobs to calculate acceleration?!? We **never** calculated acceleration! We only calculated forces.

We could instruct each "force" job to calculate acceleration by dividing by mass, F/mass = acceleration. But each job is an island of its own. **Total** acceleration is needed. Total acceleration is dependent on the sum of **all** forces acting on the ball. We could create a job responsible for gathering up all the forces, and then divide by mass to yield total acceleration.

**Figure 69 Force Gathering Job Yields Acceleration**



Justifiably, you may ask, "Why can't we simply put all the force calculations in one job, calculate acceleration right then and there and be done with it?" The short answer is, "You can!" In this example, it probably would have been simplest to do just that. However, in other cases, it may take an entire library of routines just to solve for drag. In another case, the code used to calculate lift force might be needed, yet cross force unneeded. What happens if we decide to add jet propulsion? Yet another force! In interest of modularity, scalability, and reusability of code, things are broken into pieces.

Maintaining the "Acceleration Calculator Job" is doable but could be cumbersome. Anytime a new force is added to simulation, the job has to be rewritten. Forces could arrive in multiple structures. Argument list changes would then have to be made to the "Acceleration Calculator Job". To get around all of this, a convenience was made. Trick allows you to specify what forces to sum up in the simulation definition file with a "collect statement". The "Acceleration Calculator Job" becomes a generic routine that handles all forces which are "collected". This "collection" mechanism is not limited to force summations. It is a generic way of gathering data from disparate parts of a simulation.

**Figure 70 Force Collection**

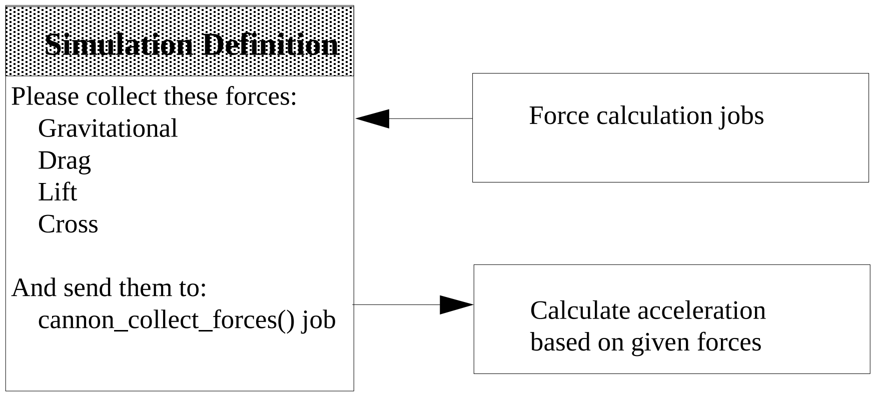


**Figure 71 Simulation Definition With Collect Statement – S_define**

```
/*****************************************************************
PURPOSE: (S_define Header)
LIBRARY_DEPENDENCY: ((cannon/aero/src/cannon_init_aero.c)
                     (cannon/aero/src/cannon_force_gravity.c)
                     (cannon/aero/src/cannon_force_drag.c)
                     (cannon/aero/src/cannon_force_lift.c)
                     (cannon/aero/src/cannon_force_cross.c)
                     (cannon/aero/src/cannon_collect_forces.c)
                     (cannon/aero/src/cannon_integ_aero.c)
                     (cannon/aero/src/cannon_impact_aero.c)
                     (cannon/aero/src/cannon_aero_default_data.c))
*****************************************************************/
#include "sim_objects/default_trick_sys.sm"
##include "cannon/aero/include/cannon_aero.h"
##include "cannon/aero/include/cannon_aero_proto.h"

class CannonSimObject : public Trick::SimObject {
   public:
      CANNON_AERO baseball ;
      Trick::Integrator* my_integ ;

      CannonSimObject() {
         ("initialization") cannon_init_aero( &baseball ) ;
         ("default_data") cannon_aero_default_data( &baseball ) ;

         ("derivative") cannon_force_gravity( &baseball ) ;
         ("derivative") cannon_force_drag( &baseball ) ;
         ("derivative") cannon_force_lift( &baseball ) ;
         ("derivative") cannon_force_cross( &baseball ) ;

         ("derivative") cannon_collect_forces( &baseball) ;
                                   Feed acceleration to cannon_integ_aero()

         ("integration") trick_ret = cannon_integ_aero(&baseball) ;

         ("dynamic_event") cannon_impact_aero( &baseball ) ;
      }
} ;

CannonSimObject dyn ;

IntegLoop dyn_integloop (0.01) dyn ;

collect dyn.baseball.force_collect = {
        dyn.baseball.force_gravity[0],
        dyn.baseball.force_drag[0],
        dyn.baseball.force_magnus[0],             Feed forces to cannon_collect_forces()
        dyn.baseball.force_cross[0] } ;
```

In the S_define, notice that the collect statement "collects" the forces into a variable called *dyn.baseball.force_collect*. This variable is the strange variable that was of type "void**" mentioned in Section 8.2.1.1. It is found in the CANNON_AERO structure.

#### 8.2.9.1 Edit S_define With Force Collection

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_aero`

**UNIX Prompt>** `vi S_define`

### 8.2.10 Collect Job

The cannon_collect_forces() job is rather strange looking. There is a macro called "NUM_COLLECT". This macro is used to access the number of variables collected in the S_define. We collected 4: force_gravity; force_drag; force_magnus; force_cross. We actually collected 4 addresses each of which pointed to the first element in each of the three element force arrays. Confusing? You will not be the first bamboozled by the collect statement! Don't worry. Check out the generic source code to calculate acceleration.

**Figure 72 Collect Job Source Code - cannon_collect_forces.c**

```
/*********************************************************
PURPOSE:    (Collect all forces and calculate acceleration)
*********************************************************/
#include "../include/cannon_aero.h"
#include "sim_services/include/collect_macros.h"

int cannon_collect_forces(CANNON_AERO *C)
{
       double **collected_forces ;
       int ii ;

       /* Collect external forces on the ball */
       collected_forces = (double**)(C->force_collect) ;
       C->force_total[0] = 0.0 ;
       C->force_total[1] = 0.0 ;
       C->force_total[2] = 0.0 ;

       for( ii = 0 ; ii < NUM_COLLECT(collected_forces) ; ii++ ) {
              C->force_total[0] += collected_forces[ii][0] ;
              C->force_total[1] += collected_forces[ii][1] ;
              C->force_total[2] += collected_forces[ii][2] ;
       }

       /* Solve for xyz acceleration */
       C->acc[0] = C->force_total[0] / C->mass ;
       C->acc[1] = C->force_total[1] / C->mass ;
       C->acc[2] = C->force_total[2] / C->mass ;

       return(0);
}
```

#### 8.2.10.1 Edit cannon_collect_forces.c

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/src`

**UNIX Prompt>** `vi cannon_collect_forces.c <edit and save>`

## 8.3 Building And Running The Simulation

### 8.3.1 Review Of All That Source Code!

In section 8.2, the simulation source code was created. Here is a list of the files:

1. cannon_aero.h
2. cannon_force_gravity.c
3. cannon_force_drag.c
4. cannon_force_lift.c
5. cannon_force_cross.c
6. cannon_init_aero.c
7. cannon_impact_aero.c
8. cannon_integ_aero.c
9. cannon_collect_forces.c
10. cannon_aero_default_data.c
11. input.py
12. S_define

## 8.3.2 Building The Simulation

If you aren't an insane perfectionist (or if you didn't copy the code), you will definitely hit some problems when trying to build. Hopefully, the error messages that are spit back to you lead you in a worthwhile direction!

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_aero`
**UNIX Prompt>** `CP`

## 8.3.3 Running The Simulation

This simulation runs in less than a second. It actually runs about as long as it takes for a ball to leave a pitcher's hand to meet a catcher's mitt. Assume that the batter missed! We will not worry about running real-time.

### 8.3.3.1 Create A Data Recording File

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_aero`
**UNIX Prompt>** `dre &`

Step 1. In the "DR Name" entry box, enter `cannon_aero`.

Step 2. In the "DR Cycle" entry box, change `0.1` to `0.01`.

Step 3. In the "Variables" pane, *double-click* `dyn`, then `baseball`, and then `pos`.
`dyn.baseball.pos[0-2]` should appear in the "References" pane. If you like, you might want to pick some other variables to view. (note: if there is an arrow next to a variable name then you click single click the arrow to expand or close the tree)

Step 4. Choose `File->Save`.
In the "Save" dialog, make the file name `cannon_aero.dr`.
Save `cannon_aero.dr` in the `Modified_data` directory

Step 5. Exit dre.

or

**UNIX Prompt>** `vi Modified_data/cannon_aero.dr <edit and save Figure 74>`

**Figure 73 cannon_aero.dr**

```
drg = trick.DRBinary("cannon_aero")

drg.add_variable("dyn.baseball.pos[0]")
drg.add_variable("dyn.baseball.pos[1]")
drg.add_variable("dyn.baseball.pos[2]")

drg.set_cycle(0.01)

drg.set_freq(trick.DR_Always)

trick.add_data_record_group(drg, trick.DR_Buffer)

drg.enable()
```

#### 8.3.3.2 Add Data Recording File To Input

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_aero/RUN_curveball`

**UNIX Prompt>** `vi input.py`

```
add --> execfile("Modified_data/cannon_aero.dr")
to the top of the input file.
```

#### 8.3.3.3 Pitch The Curveball And View Curveball Data

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_aero`

**UNIX Prompt>** `./S_main*exe RUN_curveball/input.py`

### 8.3.4 Viewing Curveball Data

To view the trajectory (both x-y and x-z), a data product's specification file (DP_baseball) needs to be created.

Step 1. Bring up the data product's application.
**UNIX Prompt>** `trick_dp &`

Step 2. *Double click* the `trick_sims` folder.

Step 3. *Double click* the `SIM_cannon_aero` folder.

Step 4. *Double click* `RUN_curveball`.

Step 5. Launch quickplot by clicking the blue lightning bolt icon.

Step 6. *Right click* `dyn.baseball.pos[0-2]` and choose expand var.

Step 7. *Double click* `dyn.baseball.pos[1]`.

Step 8. *Drag-n-drop* the `pos[0]` variable over the `"sys.exec.out.time"` variable in the Plot located in the "DP Content" pane. You will be asked to confirm the replacement. Click "Ok".

Step 9. *Single click* the "Page" icon in the "DP Content" pane.

Step 8. *Double click*, `pos[2]`. This will create another plot on the page.

Step 9. Again, *drag-n-drop* `pos[0]` onto the `"sys.exec.out.time"` variable.

Step 10. *Click* top "Plot" icon in "DP Content" pane. In the notebook change the plot title to "X -vs- Y".

Enter "X Pos" for X axis label and "Y Pos" for Y axis label. Click "Apply Change".

Step 11. *Click* the bottom "Plot". Change the title to "X -vs- Z". Enter "X Pos" for X axis label and "Z Pos" for Y axis label. *Click* "Apply Change".

Step 12. *Click* the "Page" icon. Change title to "Trajectories". *Click* "Apply Change".

Step 13. To see the plot, click the white sheet icon on the toolbar. The horizontal and vertical scales of the plot adjust automatically when you drag one or both edges of the plot.

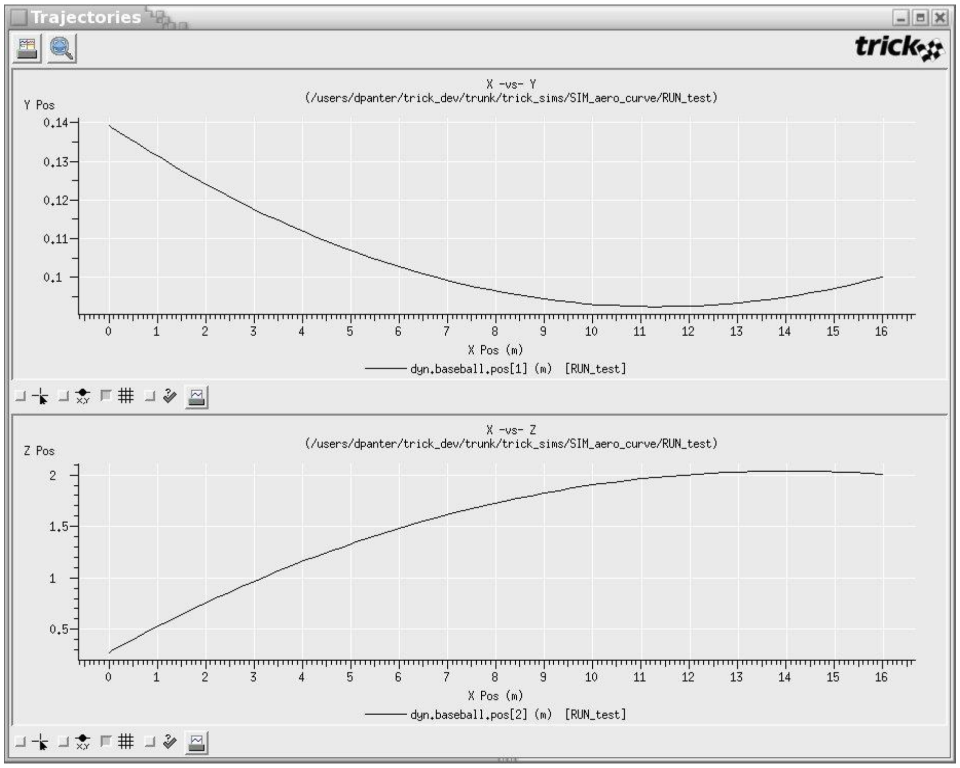
Step 13. Close the plot by closing the fxplot window.

Step 14. *Save* this as DP_baseball.
On the quickplot GUI, click File->Save As.
Click the New Folder button and type in DP_Product for the Folder name and click OK. Name the file DP_baseball. Save into the SIM_cannon_aero/DP_Product directory.

Step 15. Close the quickplot GUI.

Step 15. Close the trick_dp GUI.

**Figure 74 Curveball Trajectory**


Trajectories
trick
X -vs- Y
(/users/dpanter/trick_dev/trunk/trick_sims/SIM_aero_curve/RUN_test)
Y Pos
X Pos (m)
dyn.baseball.pos[1] (m) [RUN_test]
X -vs- Z
(/users/dpanter/trick_dev/trunk/trick_sims/SIM_aero_curve/RUN_test)
Z Pos
X Pos (m)
dyn.baseball.pos[2] (m) [RUN_test]

Note that the trajectories are X -vs- Y and X -vs- Z respectively. These are not time versus position plots.

### 8.3.5 Pitch A Fastball

A fastball occurs when the pitcher puts a back spin on the ball. To simulate this, simply change the spin velocity in the input file from 30m/s to -30m/s.

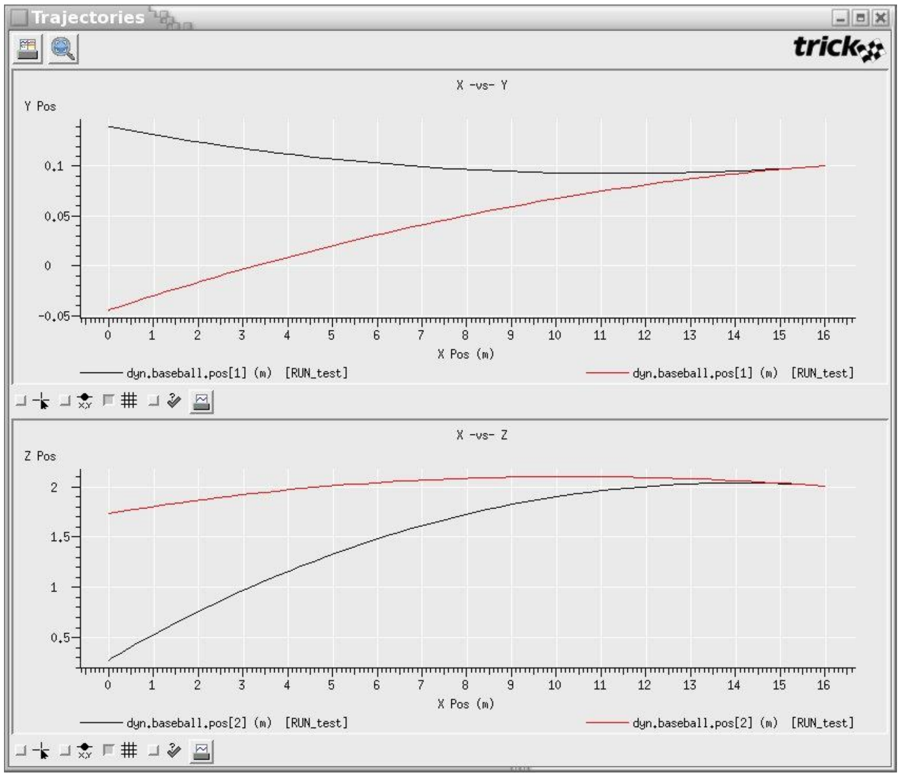
```
UNIX Prompt> cd $HOME/trick_sims/SIM_cannon_aero
UNIX Prompt> cp -r RUN_curveball RUN_fastball
UNIX Prompt> vi RUN_fastball/input.py
   Change: dyn.baseball.omega0 = trick.attach_units("rev/s",30.0)
   To:     dyn.baseball.omega0 = trick.attach_units("rev/s",-30.0)
UNIX Prompt> ./S_main*exe RUN_fastball/input.py
```

The curveball and fastball may now be compared. What a difference spin makes!

```
UNIX Prompt> cd $HOME/trick_sims/SIM_cannon_aero
UNIX Prompt> trick_dp &
```

1. *Double click* SIM_cannon_aero.
2. *Double click* RUN_curveball.
3. *Double click* RUN_fastball.
4. *Double click* DP_baseball.
5. *Click* Co-plot icon (2 collated white sheets).
6. Close the plot by closing the fxplot window.
7. Close the trick_dp GUI.

**Figure 75 Curveball -vs- Fastball**

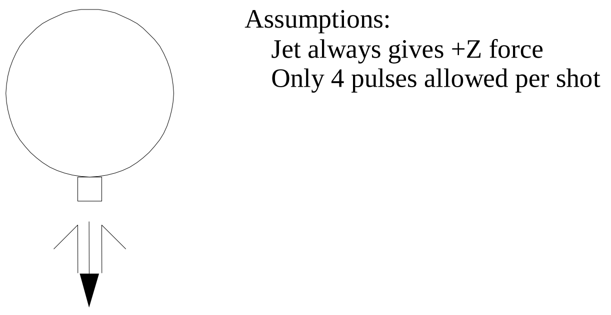


# 9.0 Adding Propulsion

Let's turn our cannonball/baseball into a guided missile.

We'll assume a very crude means of steering: one jet underneath the cannonball with a four pulse maximum. Don't ask how it stays under the ball! It just does. Each of the four jet firings gives a discrete pulse to the cannonball. The sound of the jet goes, "Phhhssstt".

**Figure 76 Propulsion**



## 9.1 Source Code

Fortunately, there isn't a lot of code to add for the control system.

### 9.1.1 Modifying The CANNON_AERO Structure

Three additions to the CANNON_AERO structure are necessary:

1. Jet fire flag
   If the jet flag is ON, the jet will fire and shut itself OFF.
2. Jet fire count
   Only four pulses allowed per shot.
3. Jet fire force
   The force fed into the derivative class job.
4. Magnitude of force in +Z direction which is configurable by user
   We will start with one ft-lb.
   The duration of the jet firing will end up being 0.1 seconds.

```
UNIX Prompt>  cd $HOME/trick_models/cannon/aero/include
UNIX Prompt>  vi cannon_aero.h
   Add the following code to the bottom of the structure:
   /* Jet */
   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_Z_plus ;   /* N Configurable force of jet in +Z dir */
```

### 9.1.2 Modifying The Default Data

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/src`

**UNIX Prompt>** `vi cannon_aero_default_data.c`

```
Add the following line:
C->force_jet_Z_plus = newton ;
```

### 9.1.3 Modifying The Prototypes

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/include`

**UNIX Prompt>** `vi cannon_aero_proto.h`

```
Add the following line:
int cannon_force_jet(CANNON_AERO*) ;
```

### 9.1.4 Job To Handle Jet Firings

This job will impart a force on the ball if the C->jet_on flag is ON... and as long as we haven't exceeded the four jet firings. The job will only fire once then shut itself off until it is directed to fire again.

**Figure 77 Jet Fire Job - cannon_force_jet.c**

```
/*************************************************
PURPOSE:      (Jet fire force)
*************************************************/
#include "../include/cannon_aero.h"

int cannon_force_jet(CANNON_AERO *C)
{
       if ( C->jet_on && C->jet_count < 4 ) {
              C->force_jet[2] = C->force_jet_Z_plus ;
              C->jet_count++ ;
              C->jet_on = 0  ;
       } else {
              C->force_jet[2] = 0.0 ;
       }
       return(0) ;
}
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/src`

**UNIX Prompt>** `vi cannon_force_jet.c <edit and save>`

### 9.1.5 Simulation Definition Changes

There are three items to add to the S_define.

First, cannon_force_jet() needs to be added to the S_define's LIBRARY_DEPENDENCIES.

Second, cannon_force_jet() needs to be added to the CannonSimObject for scheduling. Instead of classifying this as a derivative class job, the job will be classified as an "effector" class job. This is somewhat subjective, but the other forces are due to natural causes. They are part of the physical system that we don't control. We will schedule this job since we control it. We could have flight software that actually drives it. In a "control system", we would at mininum have a "sensor" system, an "effector" system, and a "brain" system which controls "effects" based on "sensed" data. This job will be scheduled every 0.1 seconds; in turn this job will also impart a force for duration of 0.1 seconds.

Third, the jet force must be added to the collect statement. The collect mechanism is already setup to sum all forces. Therefore, only a small S_define change is necessary.

```
UNIX Prompt>  cd $HOME/trick_sims
UNIX Prompt>  cp -r SIM_cannon_aero SIM_cannon_jet
UNIX Prompt>  cd SIM_cannon_jet
UNIX Prompt>  vi S_define
   Add to the header:
            (cannon/aero/src/cannon_force_jet.c)

   Add to the CannonSimObject():
            (0.1, "effector") cannon_force_jet( &baseball ) ;

   Also add new jet force to collect statement:
            collect dyn.baseball.force_collect = {
            ....
            dyn.baseball.force_jet[0] } ;
```

### 9.1.6 Running The Simulation

At this point, cannon_force_jet fires the jet when the flag "jet_on" is ON. However, we haven’t seen a way to control that input. There are a couple ways to set the "jet_on" flag:

1. Input file "read" statement.
2. Input file "event" statement.
3. We could create another job that automatically sets the "jet_on" flag when it thinks it should fire. This would imply writing a job which took on a “flight software” role.
4. We could write an external program that accepts a button push from a real live human being in the loop of the simulation.

#### 9.1.6.1 Input Read Statement

The input file’s "read" statement may be the simplest method to toggle a simulation flag at a given time. Currently, the curveball is striking the ground at home plate. Although this may be unfair and unreasonable, firing the little jet while the baseball is mid-flight will make a better pitch. The input file below is identical to the curve ball’s input file with the exception of the read statement at time=0.3 seconds. At 0.3 seconds the baseball will suddenly rise leaving the batter dumbfounded and confused. In a wild panic, he/she will swing, miss, and cry foul play!

**Figure 78 Read Statement – input.py**

```
execfile( "Modified_data/cannon_aero.dr")

dyn.baseball.pos[0] = 16.0
dyn.baseball.pos[1] = 0.1
dyn.baseball.pos[2] = 2.0

dyn.baseball.vel[0] = -30.0
dyn.baseball.vel[1] = -0.1
dyn.baseball.vel[2] = 1.0

dyn.baseball.theta = trick.attach_units("d",-90.0)
dyn.baseball.phi = trick.attach_units("d",1.0)
dyn.baseball.omega0 = trick.attach_units("rev/s",30.0)

trick.add_read(0.4, """
dyn.baseball.jet_on=1 """)

dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 6)

trick.stop(5.2)
```

```
UNIX Prompt>  mkdir RUN_jet_read
UNIX Prompt>  cd RUN_jet_read
UNIX Prompt>  cp ../RUN_curveball/input.py ./
UNIX Prompt>  vi input.py <modify and save >
```

### 9.1.6.2 Build - Run - Compare

Follow the steps below to build, run and compare the jetless curveball against the baseball with a propulsion system.

```
UNIX Prompt>  cd ../
UNIX Prompt>  make spotless
UNIX Prompt>  CP
UNIX Prompt>  ./S_main*exe RUN_jet_read/input.py
UNIX Prompt> trick_dp &
```

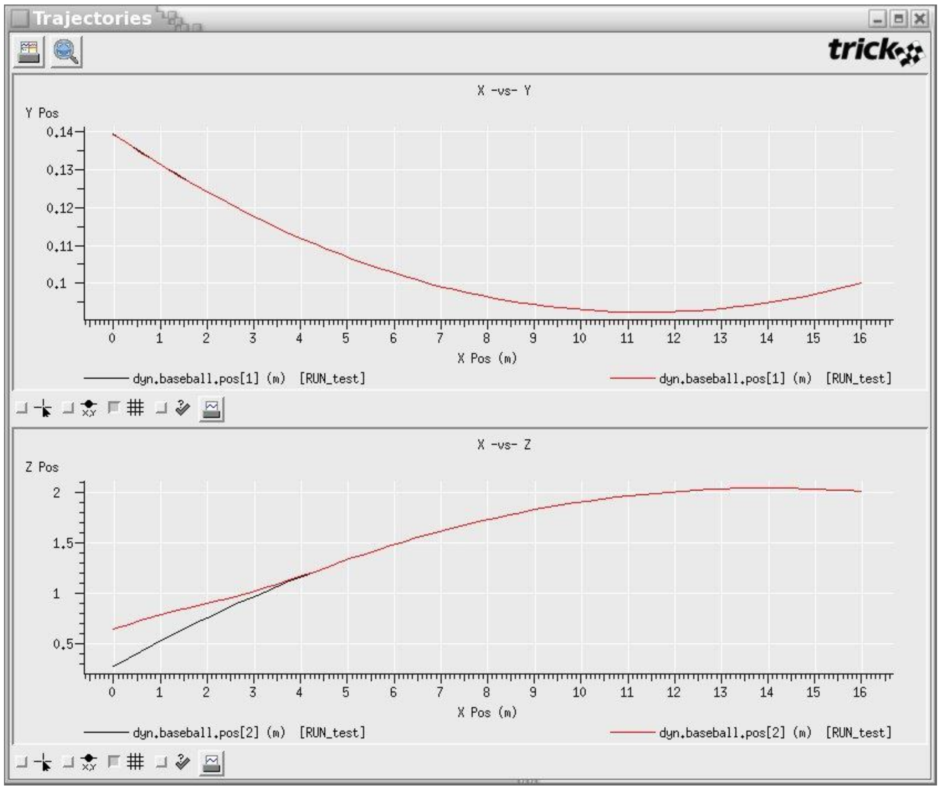
Step 1. *Double click* `SIM_cannon_jet`
Step 2. *Double click* `RUN_curveball`
Step 3. *Double click* `RUN_jet_read`
Step 4. *Double click* `DP_baseball.xml`
Step 5. *Click* the comparison plot button.

You should see something similar to the following figure.

To obtain the Y and Z axes in inches, edit DP_baseball by right clicking on DP_baseball.xml. Change the units for dyn.baseball.pos[1] and dyn.baseball.pos[2] to inches.

The following figure shows the Z tops out at 63 inches. Hmmm, 5 feet 3 inches. For the average right-handed batter, 5 feet 3 inches would hurt an awful lot for a batter peaking his head in within 5 1/2 inches of the plate. Ouch!

**Figure 79 Coplot Jet -vs- Nonjet (Y and Z are in inches)**



### 9.1.7 Input Events

What if we want to fire the jet when the baseball passes 9 meters from the plate? The following shows how an input event may be used to accomplish this. An input event is basically the combination of a condition and an action. If the condition is satisfied, the action occurs. The condition in an input event can be quite complex. Thankfully, ours is simple! Any valid input file syntax may be in the "action" of the input event. In this case, the action is to set the "jet_on" flag ON when the x position of the ball is within 9 meters of the plate. The "dt" in the input event tells how often to check the condition. In this event, dt is set to 0.01. So, every 0.01 seconds the condition will be evaluated.

**Figure 80 Input Event**

```
execfile("Modified_data/cannon_aero.dr")

dyn.baseball.pos[0] = 16.0
dyn.baseball.pos[1] = 0.1
dyn.baseball.pos[2] = 2.0

dyn.baseball.vel[0] = -30.0
dyn.baseball.vel[1] = -0.1
dyn.baseball.vel[2] = 1.0

dyn.baseball.theta = trick.attach_units("d",-90.0)
dyn.baseball.phi = trick.attach_units("d",1.0)
dyn.baseball.omega0 = trick.attach_units("rev/s",30.0)

my_event = trick.new_event("my_event")
my_event.set_cycle(0.01)
my_event.condition(0,"""dyn.baseball.pos[0] <= 9.0 """)
my_event.action(0,"""
dyn.baseball.jet_on = 1
print "action taken"
""")
trick.add_event(my_event)
my_event.activate()

dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 6)

trick.stop(1.0)
```

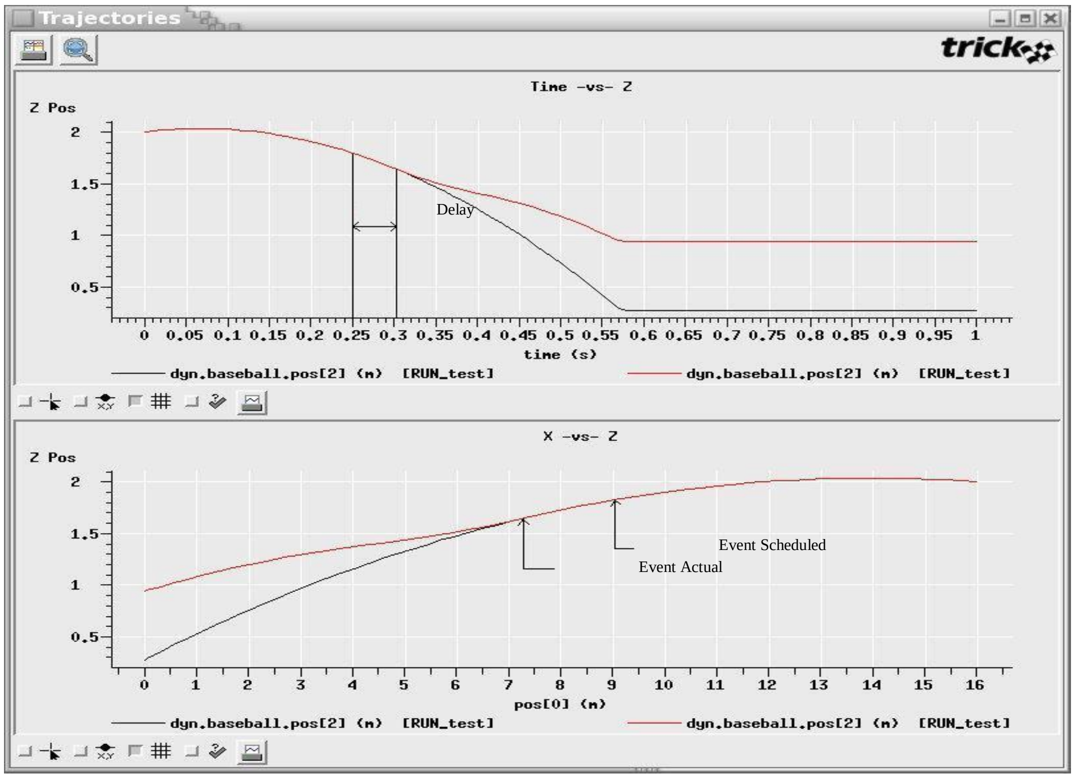
**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_jet`
**UNIX Prompt>** `mkdir RUN_jet_event`
**UNIX Prompt>** `cd RUN_jet_event`
**UNIX Prompt>** `vi input.py <edit and save>`

#### 9.1.7.1 Results

After running the simulation with the input event, you can see that the ball didn't receive an impulse until 0.3 seconds when it was 7.3247 meters from home plate. This delay is due to the fact that the "jet_on" flag was set ON at 0.25 seconds; however the job which handles the "jet_on" flag is scheduled at 0.1 seconds. So there is a delay in the jet firing from 0.25 to 0.3 seconds.

The red curve is the data associated with the event. The black curve is the data when no jet firing occurs.

**Figure 81 Delay In Jet Firing**



### 9.1.7.2 Input Event Which Corrects Delay

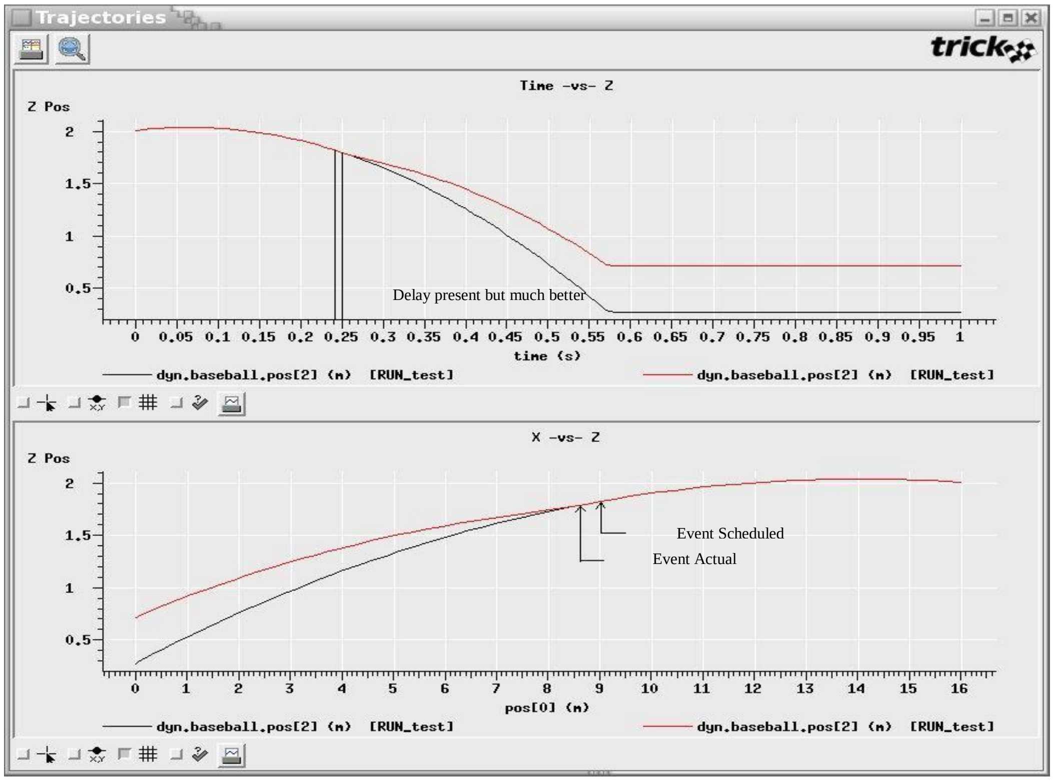
Instead of setting the "jet_on" flag ON, and suffer the delay of calling cannon_force_jet(), it is possible to call cannon_force_jet() directly from the input file. Note: You must have the job's prototype declared in a header file to call the job from the input file. If you define the prototype in the S_define, the call will not work.

See below.

```
my_event.action(0,""" dyn.baseball.jet_on = 1
trick.cannon_force_jet(dyn.baseball)
""")
```

The call statement requires “trick.” to precede the function name and the "fully qualified" name of the argument. However, the “&” must be omitted even though we are passing an address. A syntax error will be generated if the “&" is present.

**Figure 82 Input Event With Delay Correction**



This is much better. The event fired at 0.25 seconds which was much better than 0.3 seconds. However! There is still a problem. At 0.25 seconds, the ball is 8.7260 meters from the plate. Not exactly 9.0 meters! This error is due to the fact that the ball is positioned 9.0084 meters at 0.24 seconds, and 8.7260 meters at 0.25 seconds. Our input event is checking at 0.01 intervals. The resolution is 0.01. The ball slipped under the radar!

If the event must fire precisely when the ball is 9.0 meters from the plate, the simulation must look ahead in its integration stepping. Yes! You must use dynamic events for this. Input events have no notion of integration. Also, the input events aren't the most efficient. Setting the input event's dt to 0.000001 is not the way you want to do it.

## 10.0 Human In The Loop

Trick simulations have been used with mock cockpits, robotic arm hand controllers, flight displays, virtual reality head-gear, and guidance, navigation and control hardware. You may have seen an astronaut in a mock cockpit docking the space shuttle with the International Space Station. You may have seen the IMAX movie where the astronauts are training with Virtual Reality (VR). You may have gotten lucky enough to try it yourself! We haven't, hint hint. :-)

There isn't really anything too special about running Trick with a human in the loop. Trick is basically a C program. C programs can communicate using sockets, shared memory, pipes or whatever. To drive the simulation from an external source such as a human swinging his head with VR head gear, it can be as simple as scheduling a job to read the data that the head gear is registering.

## 10.1 Variable Server

Trick offers a convenient way to set/get simulation data. This is one way, among many, for a human to interact with the simulation. When a simulation runs, a TCP/IP "variable server" always runs as well. This server can be accessed to set/get simulation data. In fact, you are now familiar with three clients of the variable server: the simulation control panel; the stripchart; and the TV (Trick View) program.

As an example of using the variable server, we will write a tiny GUI for firing the jet and retrieving the X position of the cannonball.

NOTE: If you do not have access to Trick 7 tcl or if Tcl/Tk is not installed on your machine, then you will need to skip this section.

### 10.1.1 Aim High

Let's switch to shooting the ball high as we did in the first section of the tutorial. To accomplish this, the input file must be modified to configure the firing angle and initial velocity. The first time the cannon was shot it was configured at $30^o$ and 50 m/s. Equivalently, let $velocity_x = 50*\cos(30^o)$ and $velocity_z = 50*\sin(30^o)$. The input file below sets this initial velocity and puts a backspin on the cannonball. This causes the ball to rise.

**Figure 83 Re-aim Cannon - input**

```
execfile("Modified_data/cannon_aero.dr")
execfile("Modified_data/realtime_jet.py")

dyn.baseball.pos[0] = 0.0
dyn.baseball.pos[1] = 0.0
dyn.baseball.pos[2] = 0.0

dyn.baseball.vel[0] = 43.3
dyn.baseball.vel[1] = 0.0
dyn.baseball.vel[2] = 25.0

dyn.baseball.theta = trick.attach_units("d",-90.0)
dyn.baseball.phi = trick.attach_units("d",1.0)
dyn.baseball.omega0 = trick.attach_units("rev/s",30.0)

dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 6)

trick.stop(10.0)
```

```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet
UNIX Prompt>  mkdir RUN_gui
UNIX Prompt>  cd RUN_gui
UNIX Prompt>  vi input.py <edit and save>
```

### 10.1.2 Change Impact From Catcher's Mitt To Ground

The impact (contact) occurs in the catcher's mitt when simulating the curve ball. To change the impact back to the ground, it is necessary to modify cannon_impact_aero.c. Instead of using X position to brace for impact, Z position needs to be used. A historical note: On 9/08/04, JPL's Genesis solar probe crashed in the Utah desert at 100 mph after its parachutes failed to deploy. The probe was supposed to shed light on a dark spot in our knowledge concerning the origins of our solar system. We wrote this after a user found a bug in the tutorial concerning impact. In the tutorial's case, it was lack of impact.

```
UNIX Prompt>  cd $HOME/trick_models/cannon/aero/src
UNIX Prompt>  vi cannon_impact_aero.c
   Change: C->rf.error = C->pos[0];
   To    : C->rf.error = C->pos[2];
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet
UNIX Prompt>  CP
```

### 10.1.3 Make A Real-time Stripchart

To make a real-time stripchart of the trajectory, two input files needs to be setup.

**Figure 84 Real-time Input File - realtime_jet.py**

```
trick.frame_log_on ()
trick.real_time_enable ()
trick.exec_set_software_frame (0.01)
trick.exec_set_enable_freeze(True)
trick.exec_set_freeze_command(True)
trick.sim_control_panel_set_enabled(1)

trick_vs.stripchart.set_input_file("jet.sc")
```

**Figure 85 Stripchart Input File - jet.sc**

```
Stripchart:
   title = "Cannon Trajectory"
   geometry = 800x800+300+0
   x_min = 0.0
   x_max = 225.0
   y_min = 0.0
   y_max = 60.0
   x_variable = dyn.baseball.pos[0]
   dyn.baseball.pos[2]
```

```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet/Modified_data
UNIX Prompt>  vi realtime_jet.py <edit and save>
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet
UNIX Prompt>  vi jet.sc <edit and save>
```

The simulation should be ready to run now. To test it, try running it.

```
UNIX Prompt>  ./S_main*exe RUN_gui/input.py &
```

### 10.1.4 Making A GUI

How to make a GUI from scratch? For the tutorial we prefer Tcl/Tk because it is simple. You may use any language that you choose (as of Trick 10, the Trick GUIs are written in Java).
To make sure you have Tk "wish", try the following:

```
UNIX Prompt>  wish
   A little empty window should pop up.
```

The following code sample is written in Tcl/Tk. You don't have to understand it perfectly. We'll try annotation to make it semi-clear what's going on.

NOTE: Trick 13 is not packaged with the tcl code. You can use the tcl directory from a Trick 7 release just change the set auto_path line to the correct path. If you do not have access to Tcl/Tk then you may skip this section.

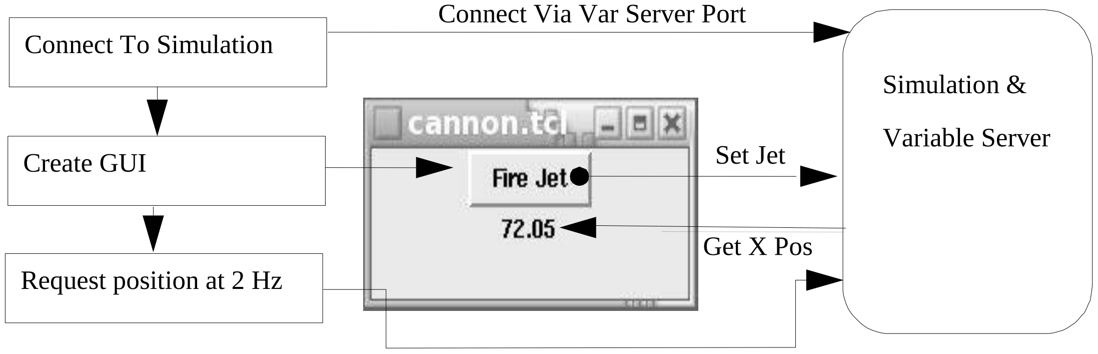
```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet
```

**UNIX Prompt>** `vi cannon.tcl (edit & save as seen in following figure)`
**UNIX Prompt>** `chmod +x cannon.tcl`

**Figure 86 Jet Fire GUI Source Code - cannon.tcl**

```
#! /bin/sh
# \
exec wish "$0" -- "$@"

# Grab package Trick's SimCom package
global auto_path
set auto_path [linsert $auto_path 0 $env(TRICK_HOME)/bin/tcl]
package require Simcom
namespace import Simcom::*

proc create { } {

       global cannon
                                                Press Button, Call "fire jet"
       button .b -text "Fire Jet" -command "fire_jet $cannon(socket)"
       label  .l -textvariable cannon(x_position)
       pack .b .l
}

proc fire_jet { sock } {                        Tell sim to fire jet
       Simcom::send_cmd $sock "dyn.baseball.jet_on = 1 ;"
}

proc get_sim_data { } {
       global cannon
                                                Request position at 2 Hz
       Simcom::send_cmd $cannon(socket)  "trick.var_cycle(0.5)\n"
       Simcom::send_cmd $cannon(socket)  "trick.var_add(\"dyn.baseball.pos\[0\]\")\n"

                                                 Get and display sim x position
       while { [gets $cannon(socket) sim_data] != -1 } {
              set sim_data [string trimright $sim_data]
              set sim_data [string range $sim_data 2 end]
              set sim_data [string trimleft $sim_data]
              set cannon(x_position) [format "%.2f" $sim_data]
              update
       }
}

proc main { } {
       global cannon
       global argv

       set cannon(port) [lindex $argv 0]            Connect to simulation
       set cannon(socket) [Simcom::connect "localhost" $cannon(port)]

       create

       get_sim_data
}
```

**Figure 87 Jet Fire GUI Flow**



## 10.1.5 Running The GUI Along With The Simulation

The simulation and the cannon.tcl mini-GUI will be run independently. First, the simulation will be started. Then the GUI will be launched. In order to start the GUI it is necessary to know what port to connect to. To find the port, a backdoor will be used.

1. Build and run simulation.

   **UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_jet`

   **UNIX Prompt>** `CP`

   **UNIX Prompt>** `./S_main*exe RUN_gui/input.py &`

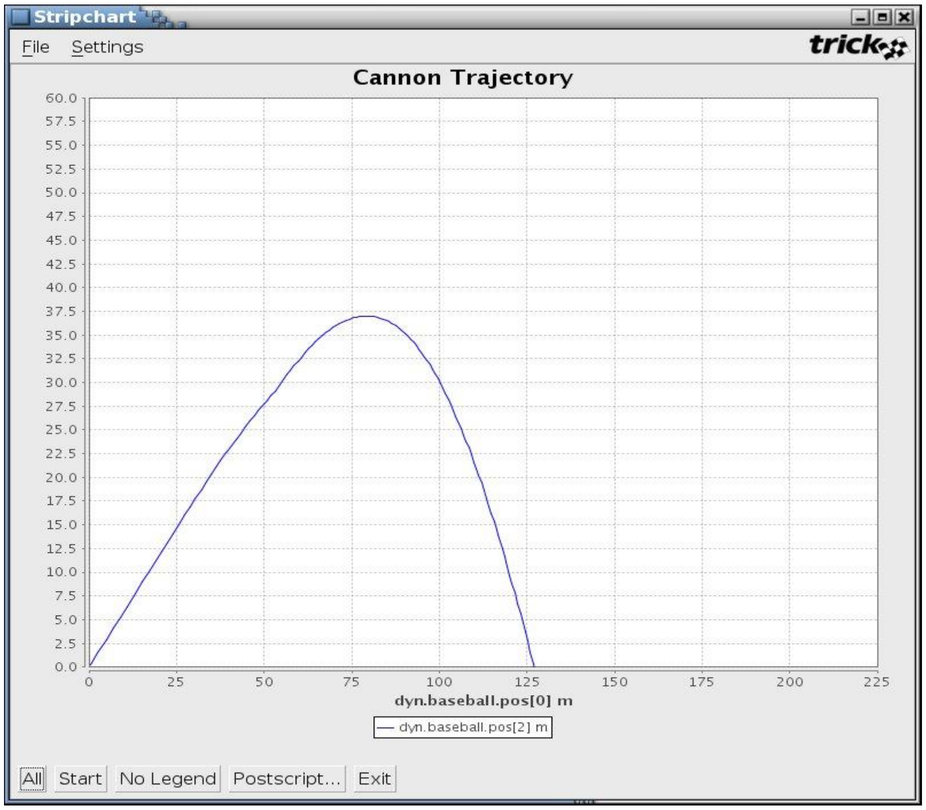
2. Locate variable server port.
   When the simulation comes up, the sim_control gui should display the simulation port below the status area at the bottom of the GUI.

3. Launch cannon.tcl with the port number found in step 2.

   **UNIX Prompt>** `./cannon.tcl <port> &`

4. Start sim and clickety-click "Fire Jet" button. Remember that there are at max 4 firings. We tried this a few times and got a max distance of 137.44 meters. If we didn't fire the jets at all, we got a distance of 123.08 meters.

**Figure 88 Firing Jets from GUI - Stripchart**



## 10.1.6 Running the sim with graphics

You may find the need to interface your Trick simulation with a graphics package. The following instructions will get you up and running with EDGE, using one of two methods to interface Trick with EDGE.

EDGE (Engineering Doug Graphics for Exploration) is the graphics toolkit normally used in conjunction with Trick simulations. It's important to note that EDGE and Trick are de-coupled. In theory, Trick could interface with any other graphics package. Since they originated in the same place, Engineering Robotics, there quite a few are tools to more seamlessly join them.

EDGE is a package containing the DOUG (Dynamic Onboard Ubiquitous Graphics) renderer, models, earth maps, scene and configuration files. DOUG is a renderer and is the underlying renderer packaged within EDGE.

Let's get started building your Trick/DOUG interface. First, you'll need to download the latest version of EDGE from the EDGE wiki. Once you've unzipped the EDGE release, you will need to set some environment variables:

```
    % setenv DOUG_HOME $HOME/EDGE_v*
    % setenv USERDATA $DOUG_HOME/userdata
```

We're now going to create a spinning cube in EDGE. Create the following two files:

1. ${USERDATA}/models/cube.ac:

**Figure 89 cube.ac**

```
AC3Db
MATERIAL "DefaultWhite" rgb 1 1 1  amb 1 1 1  emis 0 0 0  spec 0.5 0.5 0.5  shi 64  trans 0
MATERIAL "(null)" rgb 0.8 0.8 0.8 amb 0.5 0.5 0.5 emis 0 0 0 spec 1 1 1 shi 32 trans 0
OBJECT world
kids 1
OBJECT poly
name "omesh"
data 5
omesh
crease 30
numvert 8
0.5 -0.5 -0.5
0.5 -0.5 0.5
-0.5 -0.5 0.5
-0.5 -0.5 -0.5
0.5 0.5 -0.5
-0.5 0.5 -0.5
-0.5 0.5 0.5
0.5 0.5 0.5
numsurf 6
SURF 0x00
mat 1
refs 4
0 0.0 1.0
1 0.0 0.0
2 1.0 0.0
3 1.0 1.0
SURF 0x00
mat 1
refs 4
4 0.0 1.0
5 0.0 0.0
6 1.0 0.0
7 1.0 1.0
SURF 0x00
mat 1
refs 4
0 0.0 1.0
4 0.0 0.0
7 1.0 0.0
1 1.0 1.0
SURF 0x00
mat 1
refs 4
2 0.0 1.0
6 0.0 0.0
5 1.0 0.0
3 1.0 1.0
SURF 0x00
mat 1
refs 4
4 0.0 1.0
0 0.0 0.0
3 1.0 0.0
5 1.0 1.0
SURF 0x00
mat 1
refs 4
1 0.0 1.0
7 0.0 0.0
```

```
6 1.0 0.0
2 1.0 1.0
kids 0
```

2. ${USERDATA}/structs/cube.str:

**Figure 90 cube.str**

```
#Hierarchy Version 5.0
CUBE_BASE
SYSTEM
0 0 0
0 0 0
NULL
#Cameras Version 5.0
CubeCam
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
0.000000 0.000000 0.000000
NULL
#Lights Version 5.0
sun
LIGHT_DIRECT
1.000000 0.000000 0.000000
1.000000 1.000000 1.000000
NULL
sun2
LIGHT_DIRECT
-0.129000 0.063000 0.310000
1.000000 1.000000 1.000000
NULL
AMBIENT 0.0
ATTENUATION LINEAR
ATTENUATION DIST. 1000.000000
done
```

Next, we need to update two configuration files.

1. ${USERDATA}/user.cfg:

**Figure 91 user.cfg**

Copy and paste this right after DSP_CONFIG and before POSTLOAD block:

```
 LOADS
{
   CUBE
   {
      scenes
      {
         default ${USERDATA}/structs/cube.str
      }
   }
}
```

Copy and paste this in the DISPLAY block between DEFAULT and ENG_GRAPHICS:

```
CUBE
{
   channel1
   {
      view.MAIN   CubeCam perspective( 40.9, 1.333, 2.0, 1000000000000.0 ); xywh( 0, 0,
                                     640, 480 ); attribs(CAMINFO);
   }
 }
```

2. ${USERDATA}/configs/user_models.cfg:

**Figure 92 user_models.cfg**

Copy and paste the following under `DSP_CONFIG { DATA { SCENE_LOAD { MODELS {`

```
node( CUBE ); model (cube.ac); parent( CUBE_BASE );
```

We will now enable the DOUG Sim Data Plugin. This will create a drop-down menu in the EDGE gui. It will allow us to select an API file (see below) and select our spinning cube Trick simulation. In ${USERDATA} /user.cfg:

**Figure 93 user.cfg**

```
DISPLAY
{
   CUBE
   {
      plugins
      {
         Simdata   dsp_simdata
      }
   }
 }

 DOUG
 {
    GUI
    {
         Simdata   "${GUI_HOME}/simdatadlg.tcl"
    }
 }
```

We will need to map our Trick variables to DOUG Nodes. We will do this using by creating an api file. ${USERDATA}/sim_data/cube.api:

**Figure 94 cube.api**

```
CUBE xyzpyr.cube.pyr[0]   pitch  d  write  1.0
CUBE xyzpyr.cube.pyr[1]   yaw    d  write  1.0
CUBE xyzpyr.cube.pyr[2]   roll   d  write  1.0
```

Now let's make our Trick Cube simulation. You will need to create the following six files:

1. trick_sims/SIM_cube/S_define

**Figure 95 S_define**

```
/**************** TRICK HEADER ******************************
 PURPOSE:                     (S_define)
 LIBRARY_DEPENDENCY:          ((cube/src/cube_spin.c)
                               (cube/src/cube_default_data.c
 *************************************************************/

 #include "sim_objects/default_trick_sys.sm"

 ##include "cube/include/cube.h"
 ##include "cube/include/cube_proto.h"

 class CubeSimObject : public Trick::SimObject {

   public:
      CUBE cube;
      CubeSimObject() {
         ("default_data") cube_default_data( &cube );
         (0.01, "scheduled") cube_spin( &cube );
      }
 };

 CubeSimObject xyzpyr;
```

2. trick_sims/SIM_cube/ Modified_data/cube.dr

**Figure 96 cube.dr**

```
global DR_GROUP_ID global drg try:

    if DR_GROUP_ID >= 0:
        DR_GROUP_ID += 1

except NameError:

    DR_GROUP_ID = 0
    drg = []

drg.append(trick.DRAscii("cube")) drg[DR_GROUP_ID].set_freq(trick.DR_Always)
drg[DR_GROUP_ID].set_cycle(0.01) drg[DR_GROUP_ID].set_single_prec_only(False)
drg[DR_GROUP_ID].add_variable("xyzpyr.cube.pyr[0]")
drg[DR_GROUP_ID].add_variable("xyzpyr.cube.pyr[1]")
drg[DR_GROUP_ID].add_variable("xyzpyr.cube.pyr[2]")
trick.add_data_record_group(drg[DR_GROUP_ID], trick.DR_Buffer) drg[DR_GROUP_ID].enable()
```

3. trick_sims/SIM_cube/RUN_test/input.py

**Figure 97 input.py**

```
execfile("Modified_data/cube.dr")
trick.sim_control_panel_set_enabled(True)
trick.exec_set_software_frame(1e-2)
```

4. trick_models/cube/include/cube.h

**Figure 98 cube.h**

```
/**************** TRICK HEADER *************************
 PURPOSE:                     (Test cube)
 *****************************************************/

 #ifndef _cube_h_
 #define _cube_h_

 typedef struct {

        double xyz[3] ;  /* M xyz pos */
        double pyr[3] ;  /* r pitch, yaw, roll */

 } CUBE;

 #endif
```

5. trick_models/cube/src/cube_spin.c

**Figure 99 cube_spin.c**

```
/********************************** TRICK HEADER **********
 PURPOSE:             (Test cube)
 LIBRARY DEPENDENCY: ((cube_spin.o)
                      (libd_comm_tc.a)
                      (libdsp.a)
                      (dl.a))
 ***********************************************************/
 #include "../include/cube.h"
 #include "d_comm.h"

 int cube_spin ( CUBE* C ) {

   int ii;

   const double PI = 3.141592;
   double one_degree = PI/180.0;
   double two_pi = 2*PI;

   for ( ii = 0; ii < 3; ii++ ) {

      C→pyr[ii] += one_degree;

      if ( C→pyr[ii] > two_pi ) {
         C→pyr[ii] = 0.0;
      }
   }

   return 0 ;
 }
```

6. trick_models/cube/src/cube_default_data.c:

**Figure 100 cube_default_data.c**

```
/********************************** TRICK HEADER **********
 PURPOSE:             (Cube default data values)
 LIBRARY DEPENDENCY: ((cube_default_data.o)
                       (cube_init.0))
 ***********************************************************/
 #include "../include/cube.h"

 int cube_default_data( CUBE *C ) {
    C→xyz[0] = 0.0, 0.0, 0.0;
    C→pyr[0] = 0.0, 0.0, 0.0;

    return (0);
 }
```

It's time to build and run the Cube sim:

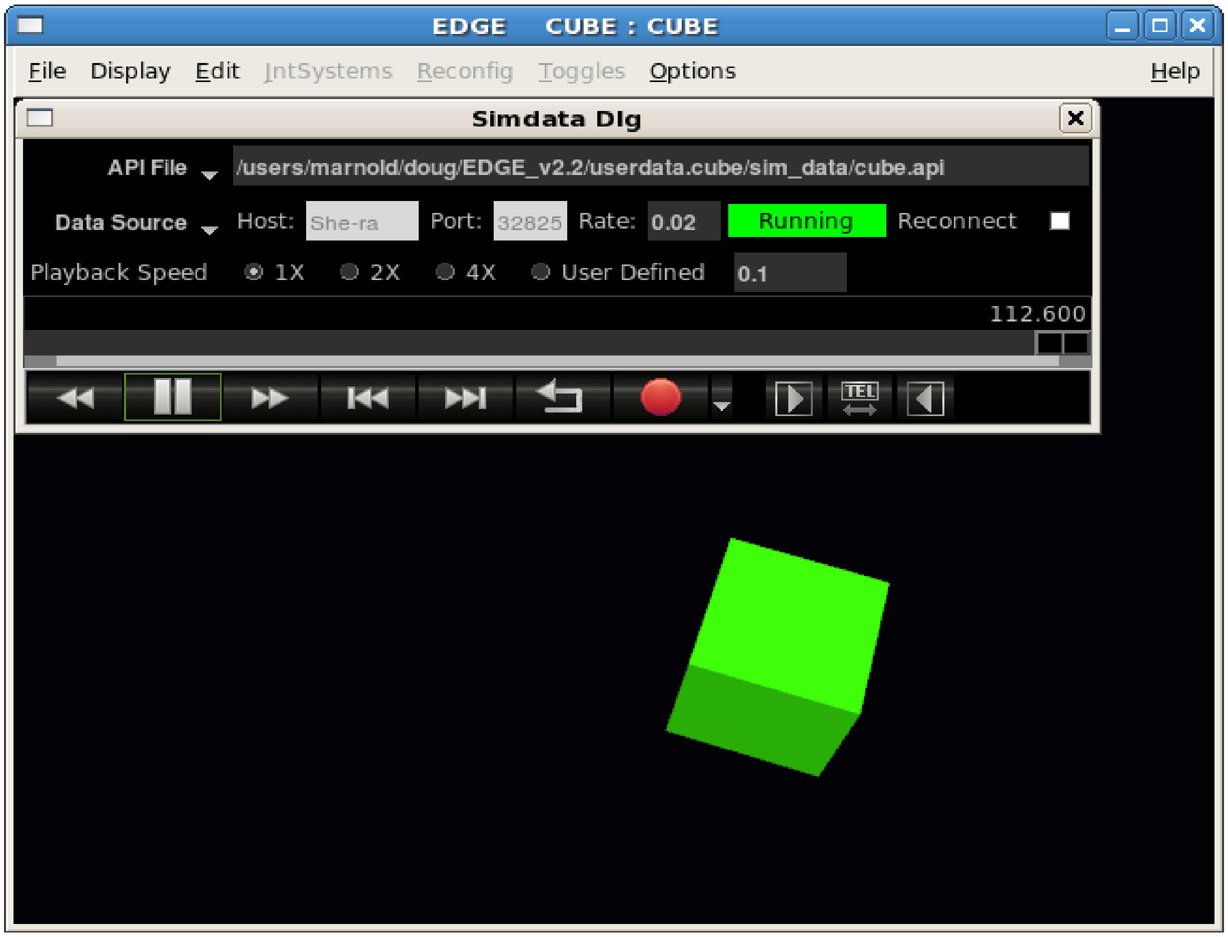
```
% cd ${TRICK_USER_HOME}/SIM_cube
% CP
% ./S_main*exe RUN_test/input
```

And now, run DOUG:

```
% cd ${DOUG_HOME}
% ./run_graphics -load CUBE -display CUBE
 * Options → Sim-data Dlg
 * Choose API File→ Browse and find the one we just made above
 * Choose Data source → Live → Browse and find the machine your sim is running on ($HOST)
```

You should get an EDGE application window displaying a spinning cube. You are looking at simulation data in real time as EDGE reads it from the Trick Variable Server.

**Figure 101 EDGE displaying spinning cube**



API File: /users/marnold/doug/EDGE_v2.2/userdata.cube/sim_data/cube.api
Host: She-ra Port: 32825 Rate: 0.02
User Defined: 0.1
112.600

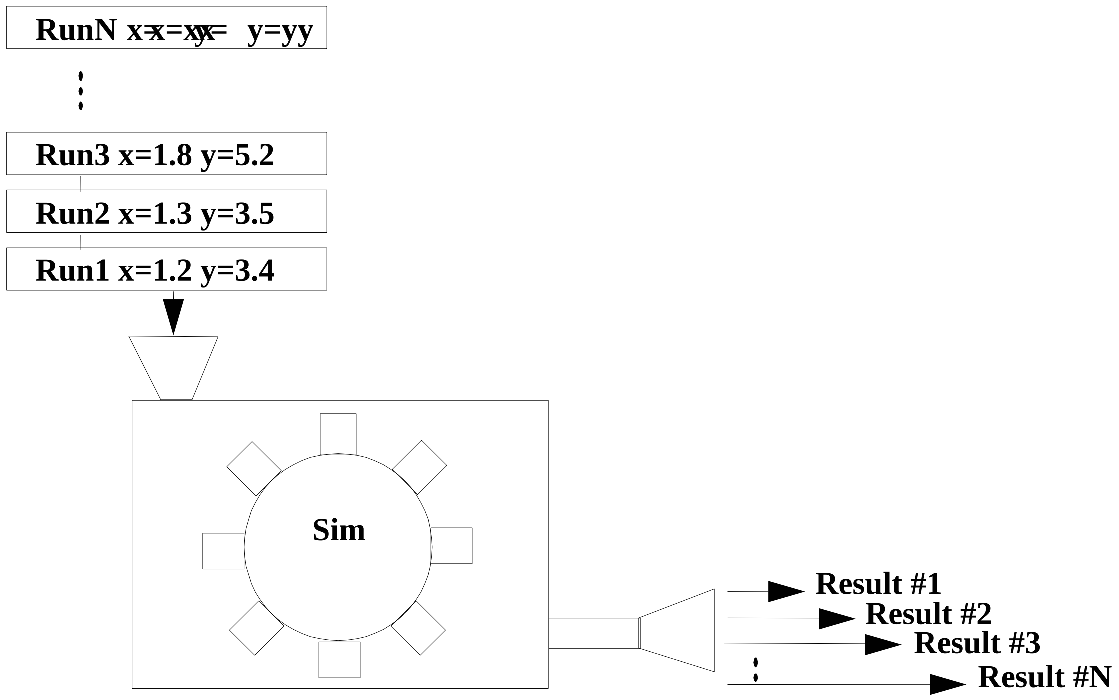
# 11.0 Monte Carlo

After shooting the cannonball and clickety-clicking the "Fire Jet" button we were able to get the cannonball to travel 152.32 meters. It'd be interesting to run a spattering of runs with different firing sequences to see which sequence gave the longest distance travelled.

Trick offers a convenient way to do just that. It gives us a simple way to run and view multiple simulation runs with varying inputs. We call this capability "Monte Carlo". We didn't invent the name "Monte Carlo". The name is actually derived from a gambling city in Monoco. It is a well known technique where mathematical problems are solved using random numbers and probability statistics. When we say "Monte Carlo", we mean running the simulation repeatedly over a varying input space. How the input space is varied is up to the user. How the input space is varied may or may not fall in the strict definition of Monte Carlo, i.e., using bell curves, linear distributions, etc.

Trick typically uses the GSL library, however, as of Trick 10.7 the library is not included in Trick releases. The user must download the GLS library and "tell" Trick in the $TRICK_HOME/configure file where to find it (add to configure file gsl=/path/to/gsl). If Trick cannot find the GSL library then it will print a message that it could not find it and will use Trick's built in random number generators. The monte carlo simulations below can still be run, but the output for the simulation using the Gaussian distribution function will have different output.

**Figure 102 Monte Carlo**



## 11.1 Monte Carlo - Example Simulation

In order to effectively run this particular simulation in a Monte Carlo style, the simulation must be instructed to fire the jet sequence through the input file. To accomplish this, four parameters are added to the CANNON_AERO structure:

time_to_fire_jet_1; time_to_fire_jet_2; time_to_fire_jet_3 and time_to_fire_jet_4. A job will be created that fires the jets if the simulation time is equal to any of the "time_to_fire_jet" variables.

## 11.2 Modify CANNON_AERO structure

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/include`

**UNIX Prompt>** `vi cannon_aero.h`

```
Add the following to the CANNON_AERO structure:
double time_to_fire_jet_1 ;    /* s First jet fire time */
double time_to_fire_jet_2 ;    /* s Second jet fire time */
double time_to_fire_jet_3 ;    /* s Third jet fire time */
double time_to_fire_jet_4 ;    /* s Fourth jet fire time */
```

**UNIX Prompt>** `vi cannon_aero_proto.h`

```
Add the following to the prototypes:
int cannon_jet_control(CANNON_AERO*);
```

## 11.3 Create New Job For Firing Jets At Given Times

**Figure 103 Controlled Jet Firings - cannon_jet_control.c**

```
/************************************************************
PURPOSE:         (Controls when jets are fired)
************************************************************/
#include "../include/cannon_aero.h"
#include "sim_services/Executive/include/exec_proto.h"
#include "trick_utils/math/include/trick_math.h"
#define CANNON_EQUALS(X,Y) ( fabs(X  - Y) < 1.0e-9 ) ? 1 : 0

int cannon_jet_control( CANNON_AERO* C )
{
        double sim_time ;

        sim_time = exec_get_sim_time() ;    Built-in Trick Function For Accessing Sim Time

        if ( CANNON_EQUALS(sim_time, roundoff(0.1, C->time_to_fire_jet_1)) ||
             CANNON_EQUALS(sim_time, roundoff(0.1, C->time_to_fire_jet_2)) ||
             CANNON_EQUALS(sim_time, roundoff(0.1, C->time_to_fire_jet_3)) ||
             CANNON_EQUALS(sim_time, roundoff(0.1, C->time_to_fire_jet_4)) ) {

                C->jet_on = 1 ;
        }
        return(0) ;
}
```

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/src`

**UNIX Prompt>** `vi cannon_jet_control.c <edit and save>`

## 11.4 Add Jet Control Job To S_define

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_jet`

**UNIX Prompt>** `vi S_define`

```
 Add the following to the LIBRARY DEPENDENCIES list
      (cannon/aero/src/cannon_jet_control.c)
```

```
Add the following code after the "cannon_impact_aero()" job.  Notice that
the job is a "scheduled" job.  Scheduled jobs run before effector jobs.
The "C->jet_on" flag will always be fed into the cannon_force_jet() job.

      (0.1, “scheduled”) cannon_jet_control( &baseball ) ;
```

## 11.5 Modify The Default Data (More Boost)

To see the effects of the jet a little better, double the force of the jet firing.

**UNIX Prompt>** `cd $HOME/trick_models/cannon/aero/src`

**UNIX Prompt>** `vi cannon_aero_default_data.c`

```
   Change:  C->force_jet_Z_plus = 1.0 * newton ;
   To:      C->force_jet_Z_plus = 2.0 * newton ;
```

## 11.6 A Test Input File

**Figure 104 Test Input File For Jet Firings – input.py**

```
execfile("Modified_data/cannon_aero.dr")

dyn.baseball.pos[0] = 0.0
dyn.baseball.pos[1] = 0.0
dyn.baseball.pos[2] = 0.0

dyn.baseball.vel[0] = 43.3
dyn.baseball.vel[1] = 0.0
dyn.baseball.vel[2] = 25.0

dyn.baseball.theta = trick.attach_units("d",-90.0)
dyn.baseball.phi = trick.attach_units("d",1.0)
dyn.baseball.omega0 = trick.attach_units("rev/s",30.0)

dyn.baseball.time_to_fire_jet_1 = 3.0
dyn.baseball.time_to_fire_jet_2 = 4.0
dyn.baseball.time_to_fire_jet_3 = 5.0
dyn.baseball.time_to_fire_jet_4 = 6.0

dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 6)

trick.stop(10.0)
```

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_jet`

**UNIX Prompt>** `mkdir RUN_timed_jets`

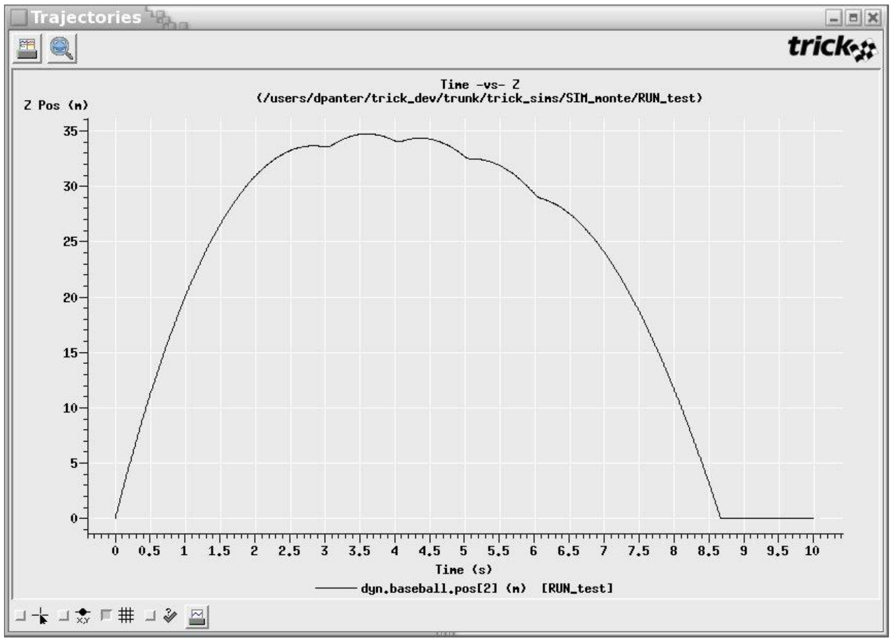
**UNIX Prompt>** `cd RUN_timed_jets`

**UNIX Prompt>** `vi input.py <edit and save>`

## 11.7 Running The Simulation With Test Input File

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_jet`
**UNIX Prompt>** `make spotless`
**UNIX Prompt>** `CP`
**UNIX Prompt>** `./S_main*exe RUN_timed_jets/input.py`

**Figure 105 Jets Firing At 3-4-5-6 Seconds**



## 11.8 Hard Coded Monte Carlo Input File

In order to run a Monte Carlo simulation with hard-coded input values it is necessary to create a Monte Carlo input file. By Trick convention, Monte Carlo input filenames are prefixed with "M_". The Monte Carlo input file shown below sets up 8 runs with various jet-firing sequences.

**Figure 106 Monte Carlo Input - M_jet_firings_inline**

```
0  1.0000   1.5000   2.0000   2.5000
1  1.5000   2.0000   2.5000   3.0000
2  2.0000   2.5000   3.0000   3.5000
3  2.5000   3.0000   3.5000   4.0000
4  3.0000   3.5000   4.0000   4.5000
5  3.5000   4.0000   4.5000   5.0000
6  4.0000   4.5000   5.0000   5.5000
7  4.5000   5.0000   5.5000   6.0000
```

A list of values for variables. The data is organized via columns per variable. The second column lists the values for one variable, the third column list values for a second variable, etc.

### 11.8.1 Create M_jet_firings_inline

```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet
UNIX Prompt>  vi M_jet_firings_inline <edit and save>
```

## 11.9 Simulation Input File

The simulation input file needs to know where the Monte Carlo input file is located. Also, Monte Carlo must be activated.

**Figure 107 Adding Monte Carlo To Input File – input.py**

```
execfile("Modified_data/cannon_aero.dr")

dyn.baseball.pos[0] = 0.0
dyn.baseball.pos[1] = 0.0
dyn.baseball.pos[2] = 0.0

dyn.baseball.vel[0] = 43.30
dyn.baseball.vel[1] = 0.0
dyn.baseball.vel[2] = 25.0

dyn.baseball.theta  = trick.attach_units("d" , -90.0)
dyn.baseball.phi    = trick.attach_units("d" , 1.0)
dyn.baseball.omega0 = trick.attach_units("rev/s" , 30.0)

dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 6)
trick.mc_set_enabled(1)
trick.mc_set_num_runs(8)

var0 = trick.MonteVarFile("dyn.baseball.time_to_fire_jet_1","M_jet_firings_inline", 2)
var1 = trick.MonteVarFile("dyn.baseball.time_to_fire_jet_2","M_jet_firings_inline", 3,"s")
var2 = trick.MonteVarFile("dyn.baseball.time_to_fire_jet_3","M_jet_firings_inline", 4)
var3 = trick.MonteVarFile("dyn.baseball.time_to_fire_jet_4","M_jet_firings_inline", 5,"s")

trick_mc.mc.add_variable(var0)
trick_mc.mc.add_variable(var1)
trick_mc.mc.add_variable(var2)
trick_mc.mc.add_variable(var3)
trick.stop(10.0)
```

- **trick.mc_set_enabled(1):** This enables the monte carlo feature.
- **trick.mc_set_num_runs(8):** This tells how many runs to fire off. We will be running with 8 different values.
- **var0 = trick.MonteVarFile("dyn.baseball.time_to_fire_jet_1", "M_jet_firings_inline", 2):** This line sets up variables that we may choose to vary their values and in the manner that we wish to calculate the values (this example is hard-coded values). It saves the name of a variable that we wish to vary its value per run "dyn.baseball.time_to_fire_jet_1". The second parameter saves the name of the file to locate the values to use for this variable. And the third parameter saves which column from the input file to use for this variables values.

- **trick_sys.sched.add_variable(var0):** This tells Trick to vary the data for the variable information on the "var0=" line. You must include this line for each variable for Trick to actually vary the variables values.

```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet
UNIX Prompt>  mkdir RUN_monte
UNIX Prompt>  cd RUN_monte
UNIX Prompt>  vi input.py <edit and save>
```

## 11.10 Secure Shell (ssh)

Before running Monte Carlo, you must have secure shell (ssh) working. If you are interested in why this is the case, Monte Carlo is designed to run distributed on several machines. Underneath the hood, even when it runs on one machine, it runs in a distributed fashion. To run distributed, it is necessary to login into other computers. ssh was the default pick for logging into other machines. If you do not have ssh (which is a shame!), you may also use rsh.

To test whether you are setup to use ssh, try the following:

```
UNIX Prompt>  ssh localhost ls
   ssh may gripe
      The authenticity of host 'localhost'...
      Are you sure you want to continue connecting (yes/no)?
   Say "yes".

   Then it may ask for your password.  Type in your password.
   (There is a way to setup ssh so that it will not ask for a password).

   Once it accepts the password, it will issue the "ls" command.
```

### 11.10.1 Running The Simulation

Running the Monte Carlo simulation is no different than running a regular single pass simulation.

```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet
UNIX Prompt>  ./S_main*exe RUN_monte/input.py
   Don't put it in background with "&" since it will more than likely ask for
   password.  If it does ask for a password, enter it to satisfy ssh.
UNIX Prompt>  ls MONTE_RUN_monte
   You should see "monte_runs", "monte_header", "run_summary" and "RUN_00000
   - RUN_00007"
```

## 11.11 Viewing Data

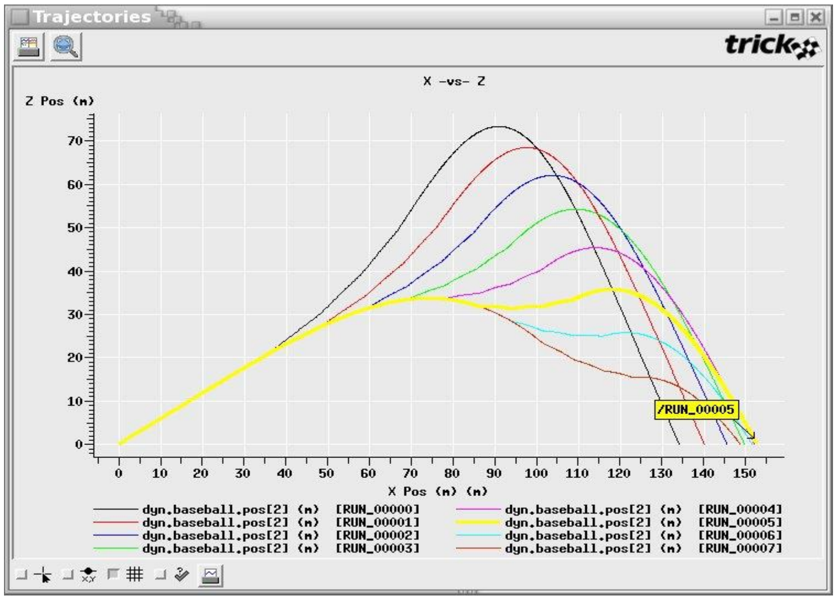
To get the best view of the trajectories generated by the Monte Carlo run, it is best to create a DP file that only has X-vs-Z. The current DP_baseball file contains X-vs-Y and X-vs-Z. We only need to delete the X-vs-Y plot. The data products program handles directories that begin with MONTE differently. If a MONTE* directory is chosen, it will collate all the curves from the individual Monte Carlo runs. To see this in action follow the steps below.

1. Bring up Trick's data products:

```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet
UNIX Prompt>  trick_dp &
```

2. *Double click* "`SIM_cannon_jet`".
3. *Highlight* "`DP_baseball.xml`" in the "DP Content" pane on the right.
4. Then *right click* --- select "Edit DP" from the menu.
5. The qp GUI should pop-up.
6. *Select* Page in the DP Content window
7. *Select* the "X -vs- Y" plot.
8. *Right click* the selection, and choose "Remove".
9. Choose File->"Save As".
10. Save as SIM_cannon_jet/DP_Product/DP_xz.
11. Exit the qp GUI.
12. In the trick_dp, click Session->Refresh and the DP_xz.xml should appear in the DP Tree window.
13. *Right click* the "`MONTE_RUN_monte`" folder under the "`SIM_cannon_jet`" folder and select "Add run(s)".
14. *Double click* "`DP_xz.xml`" in the "DP Content" pane.
15. *Click* the "Co-plot" plot icon-button (icon looks like 2 overlapped white sheets).

**Figure 108 Monte Carlo Inline Data Plot**


Trajectories
trick
X -vs- Z
Z Pos (m)
X Pos (m) (m)
/RUN_00005
dyn.baseball.pos[2] (m) [RUN_00000]
dyn.baseball.pos[2] (m) [RUN_00001]
dyn.baseball.pos[2] (m) [RUN_00002]
dyn.baseball.pos[2] (m) [RUN_00003]
dyn.baseball.pos[2] (m) [RUN_00004]
dyn.baseball.pos[2] (m) [RUN_00005]
dyn.baseball.pos[2] (m) [RUN_00006]
dyn.baseball.pos[2] (m) [RUN_00007]

Notice the yellow label with "RUN_00005" in it. To be able to tell what curve goes with what run, you may hold down the "Shift" key and then use the mouse and left-click on the curve of interest.

In our case "RUN_00005" made it the furthest. What is RUN_00005? To find out follow the steps below.

**UNIX Prompt>** `cd $HOME/trick_sims/SIM_cannon_jet/MONTE_RUN_monte`
**UNIX Prompt>** `vi monte_runs`

In "monte_runs" you will see a dataline with ID "00005" in the DATA section. The looks like:

```
00005  3.5000   4.0000   4.5000   5.0000
```

This means that of our 8 runs, the 5th run firing the jets at 3.5, 4.0, 4.5 and 5.0 seconds gives the furthest shot of what is about 152.75 meters.

## 11.12 Running A 500 Random Runs

Trick offers a couple ways to create randomized inputs for the simulation runs. In the following input file, the variables are added using a trick.MonteVarRandom method so the data is created by formulas. In this particular case, a Gaussian distribution is used for each variable.

**Figure 109 Input Using Gaussian Distribution-M_jet_firings_gaussian**

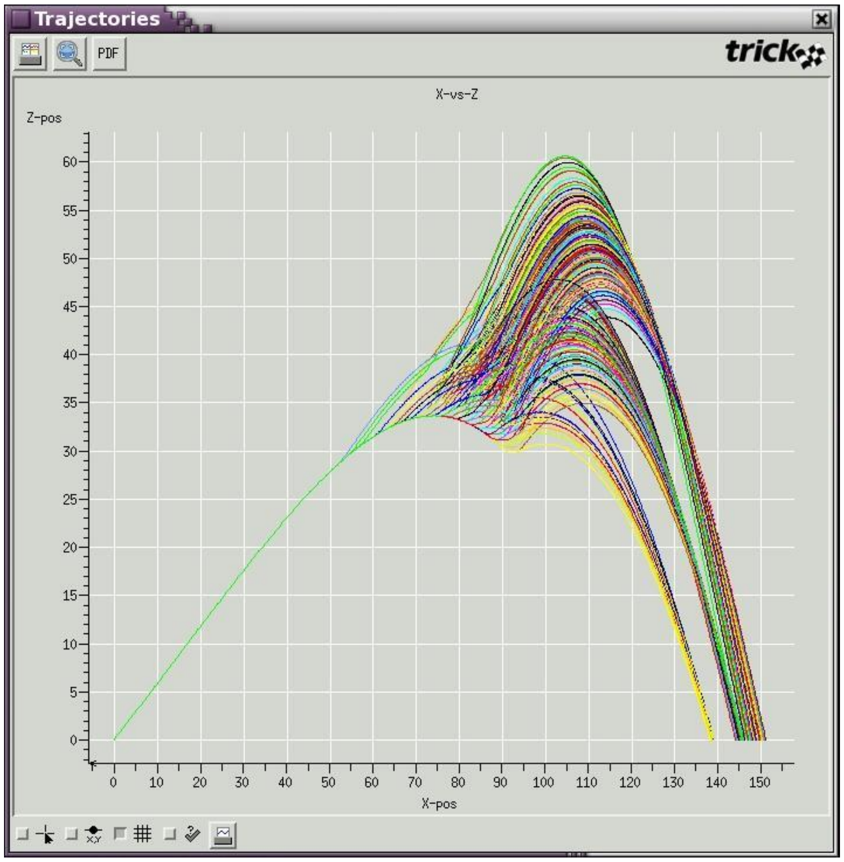
```
execfile("Modified_data/cannon_aero.dr")

dyn.baseball.pos[0] = 0.0
dyn.baseball.pos[1] = 0.0
dyn.baseball.pos[2] = 0.0

dyn.baseball.vel[0] = 43.30
dyn.baseball.vel[1] = 0.0
dyn.baseball.vel[2] = 25.0

dyn.baseball.theta  = trick.attach_units("d" , -90.0)
dyn.baseball.phi    = trick.attach_units("d" , 1.0)
dyn.baseball.omega0 = trick.attach_units("rev/s" , 30.0)
```

```
trick.mc_set_enabled(1)
trick.mc_set_num_runs(500)

var0=trick.MonteVarRandom("dyn.baseball.time_to_fire_jet_1", trick.MonteVarRandom.GAUSSIAN)
var0.set_seed(1)
var0.set_sigma(0.6667)
var0.set_mu(4.0)
var0.set_min(-4.0)
var0.set_max(4.0)

var1=trick.MonteVarRandom("dyn.baseball.time_to_fire_jet_2", trick.MonteVarRandom.GAUSSIAN)
var1.set_seed(2)
var1.set_sigma(0.6667)
var1.set_mu(4.0)
var1.set_min(-4.0)
var1.set_max(4.0)

var2=trick.MonteVarRandom("dyn.baseball.time_to_fire_jet_3", trick.MonteVarRandom.GAUSSIAN)
var2.set_seed(3)
var2.set_sigma(0.6667)
var2.set_mu(4.0)
var2.set_min(-4.0)
var2.set_max(4.0)

var3=trick.MonteVarRandom("dyn.baseball.time_to_fire_jet_4", trick.MonteVarRandom.GAUSSIAN)
var3.set_seed(4)
var3.set_sigma(0.6667)
var3.set_mu(4.0)
var3.set_min(-4.0)
var3.set_max(4.0)

trick_mc.mc.add_variable(var0)
trick_mc.mc.add_variable(var1)
trick_mc.mc.add_variable(var2)
trick_mc.mc.add_variable(var3)
```

```
dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 6)

trick.stop(10.0)
```

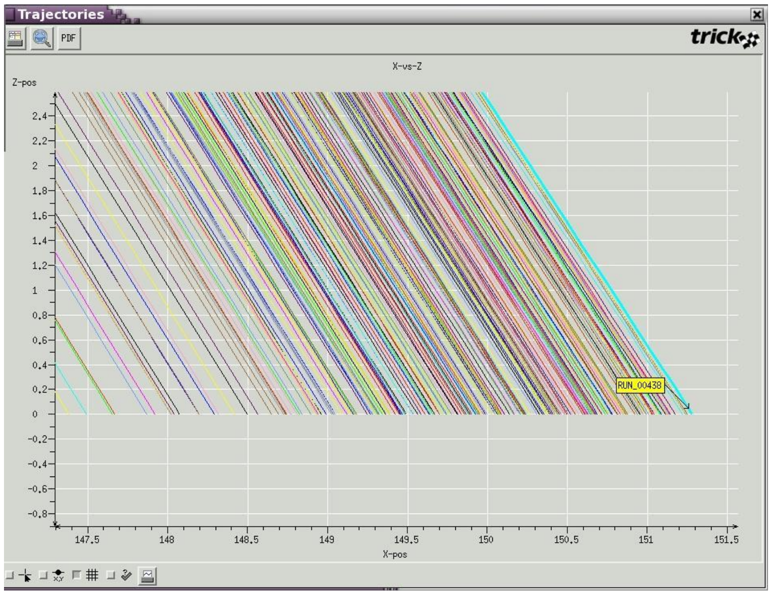
```
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet/RUN_monte
UNIX Prompt>  vi input.py
UNIX Prompt>  cd $HOME/trick_sims/SIM_cannon_jet
UNIX Prompt>  ./S_main*exe RUN_monte/input.py
```

**Figure 110 Monte Carlo Gaussian Data Plot**



There are 500 curves on that plot! Curves are colored in bands and seem randomized since we are using random data.

**Figure 111 Monte Carlo Gaussian Zoom**



```
        RUN 438

Jet 1>  3.3 seconds
Jet 2>  3.8 seconds
Jet 3>  4.4 seconds
Jet 4>  5.0 seconds

A whopping 152.79 meters
```

By zooming through the rainbow and "Shift-Clicking" the most extreme curve, run 468 is found to be the most extreme. The data shown on the right of the plot above was gathered from The sequence is not in order, but no matter, it occurs in order because of the way cannon_jet_control_monte.c was written.

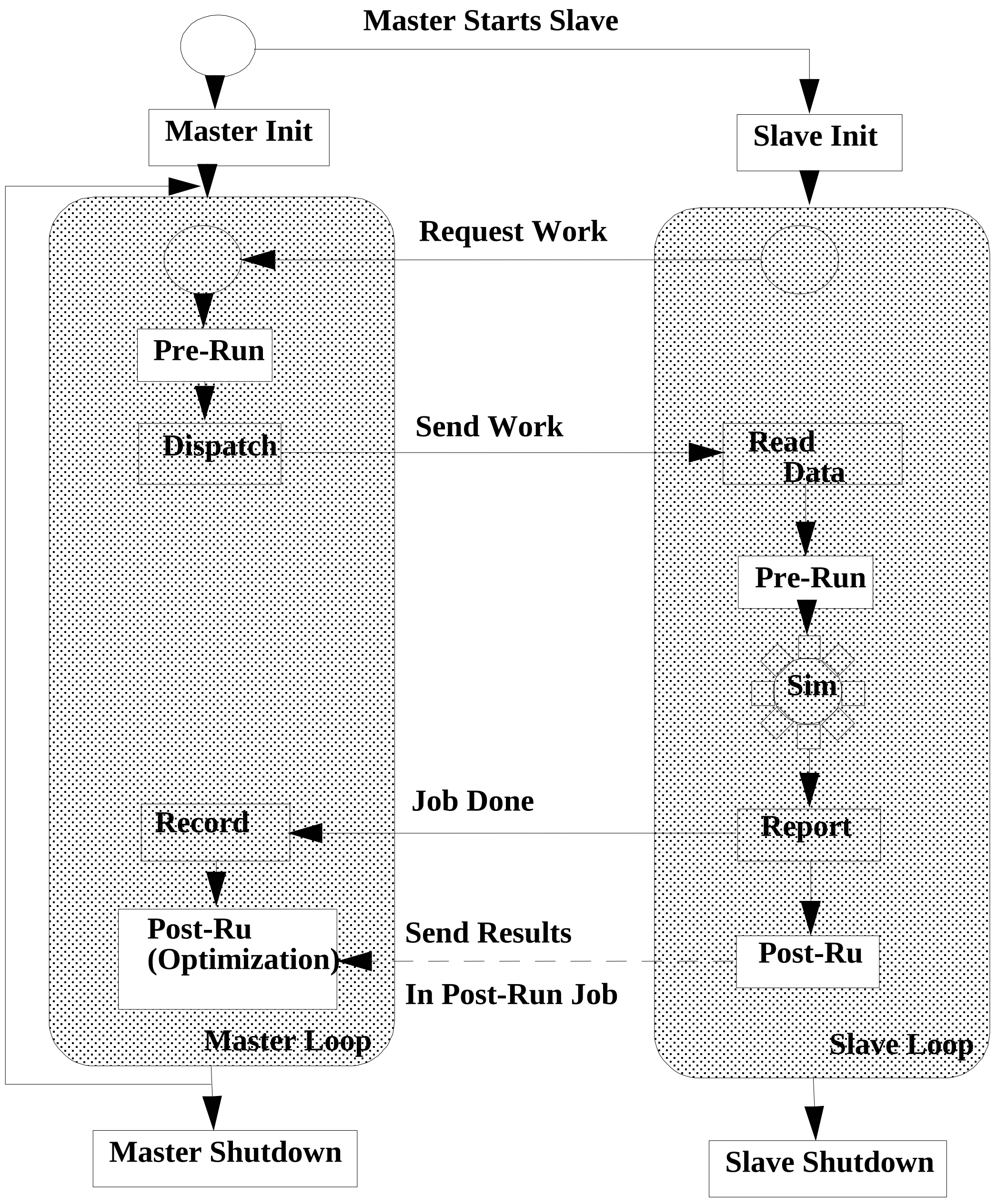
**Figure 112 Excerpt From "monte_runs" File**

```
00463       4.495810504       4.453155982       3.346320724       3.287949724
00464       4.303575158       4.755921827       3.885017788       4.310771789
00465       4.158178568       4.236484914       4.398467191       3.626823708
00466       4.103044735       4.498103238       4.60346435        3.252340614
00467       3.50168522        4.162120494       5.279005953       3.963999618
00468       3.829396855       3.26858157        5.005660295       4.420482756
00469       3.961623587       3.793083297       4.07486172        3.778977447
00470       5.339574541       4.209506936       3.502728684       2.57410264
00471       3.493228611       4.565292793       4.087798171       4.279734674
00472       3.794740284       2.799034453       3.770648065       4.029548506
00473       3.792467342       2.985653741       4.408343018       4.57128626
```

# 12.0 Optimization

Instead of using the semi-brute force technique of Monte Carlo to find the best firing sequence, an optimal solution might be found by iteratively analyzing the simulation results and intelligently deciding the next set of inputs. If there are enough runs and the decision making follows an optimal path, a best solution can be found. As of Trick 10.0, there are no canned optimization methods; however, a framework for optimization is available. Trick offers the framework for modifying inputs on the fly based on past simulation results. The degree of intelligence, or optimization method used, is up to you. In the future, we do hope to add optimization methods to Trick core. This is a first step in what promises to be an exciting path (hopefully an optimal one). :-)

**Figure 113 Master/Slave And Optimization**



In order to take advantage of optimization (feeding the simulation inputs based on output results), it is necessary to understand a little of the internals. Trick has implemented Monte Carlo/optimization with a "master slave" model. There is a "master" who creates "slaves" and tasks them to run the sim with given inputs. Each slave is a fully functional independent Trick simulation. Slaves communicate with the master over a TCP/IP socket (even if on the same machine). The master receives a request from a slave. The bored slave requests work. Once the master receives such a request, it runs user-defined pre-run jobs (if there are any). It then dispatches a packet of information to the requesting slave. The packet contains initialization data for the slave to run. After the slave receives the initialization data, it runs its own user defined pre-run jobs. After this, the slave cranks the initialization data through the simulation. Once the slave's simulation run terminates, the slave reports back to the master. The master cheers the slave and marks this job off of its list of things to do. The slave then runs its post-run jobs. In the case of optimization the slave's post-run job is very important. This is where the slave's simulation results are passed back to the master. They are passed back through a socket using a tc_write(). This write is within the post-run job itself. The master will receive the results with a tc_read() in its own post-run job. Once the master receives the results, it can make a decision on which direction it would like to go to further optimize the problem it is working on. The cycle repeats itself as often as necessary. Before the master and slave loops, any user-defined "init" jobs run. After the loops, any user defined "shutdown" jobs run.

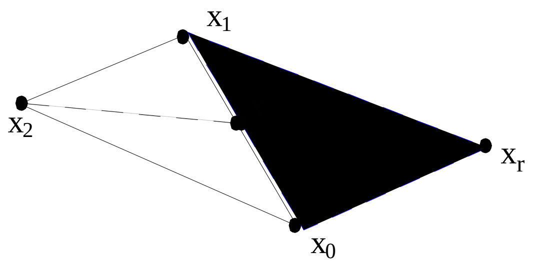
There may be several slaves running on several machines. To take advantage of distribution, there are some minor additions to the input file that have to be made. We will not delve into that area. See User's Guide for more information.

## 12.1 Amoeba Algorithm (Nelder-Meade Simplex Algorithm)

For the purposes of showing how to implement optimization within Trick, the Amoeba algorithm was chosen (also called the Nelder-Meade Simplex Algorithm). The code is original. Which means it needs mileage! The algorithm itself is very standard and can even be found in Numerical Recipes (btw, the Numerical Recipe bunch disallows use of their source code). The algorithm was found at http://www.research.ibm.com/infoecon/paps/html/amec99_bundle/node8.html.

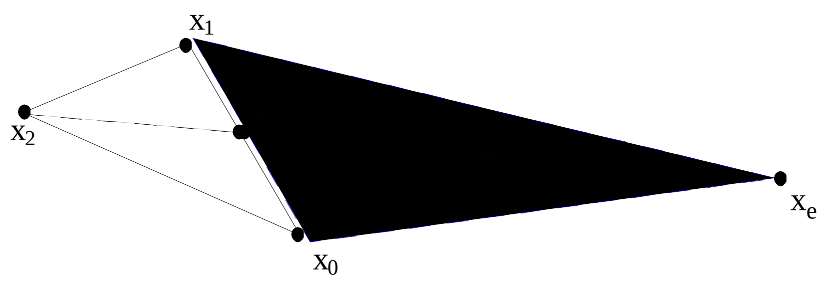
The algorithm optimizes a function $F(x)$ where $x \in \Re^n$. To begin, an initial guess is made. The guess is *n+1* points, $(x_0, x_1, \dots x_n)$ which forms an "amoeba" with *n+1* vertices. In this case, there are four jet firings. The size of the amoeba will be 5 points in the 4th dimension. Just imagine an amoeba crawling its way through the fourth dimension! $F(x)$ is the simulation itself. We've already seen $F(3.3, 3.8, 4.4, 5.0) = 152.79$ (the times of the four jet firings, yield a final distance of 152.79 meters traveled). The initial guess has a bearing on how the amoeba crawls. If it is given a bad start, the poor amoeba may crawl in the wrong direction and give poor results. The initial guess needs to be educated and is an art in itself. Once the guess is made, the amoeba is on its own. The amoeba has four possible movements: reflection, expansion, contraction and shrinking. The movement sequence or iteration is dependent on the success of the function evaluations after each move. The following diagrams show the possible movements where $x_i \in \Re^2$. Assume in the following diagrams that $F(x_0) \geq F(x_1) \geq F(x_2)$. Note that $x_2$ is then the worst point of the amoeba, in general $x_n$ is the worst point.

**Figure 114 Amoeba Reflection**



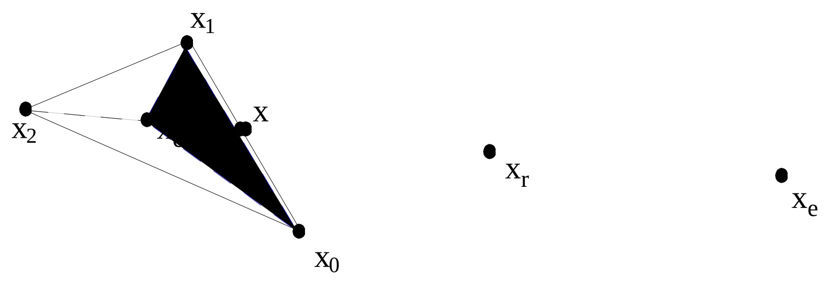
x is the "centroid" of ($x_0$, $x_1$). In general $\bar{x} = \left(\frac{1}{n}\right)\left(\sum_{i=0}^{n-1} x_i\right)$. Notice that the worst point is not taken into account for the centroid calculation. The point $x_r$ is the "reflection point", $x_r = 2\bar{x} - x_n$.

**Figure 115 Amoeba Expansion**



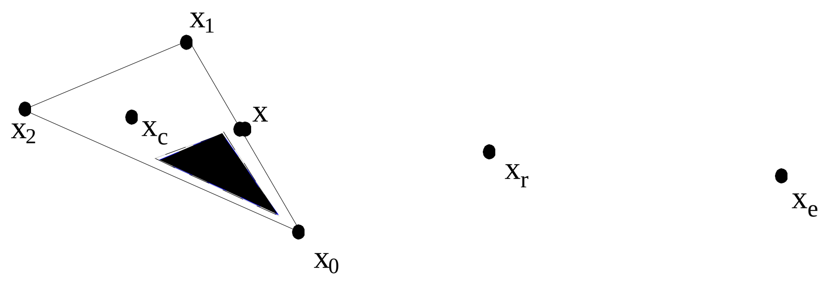
$x_e$ is the "expansion point", $x_e = 2x_r - \bar{x}$.

**Figure 116 Amoeba Contraction**



$x_c$ is the "contraction point", $x_c = \bar{x} + \frac{1}{2}(\bar{x} - x_n)$.

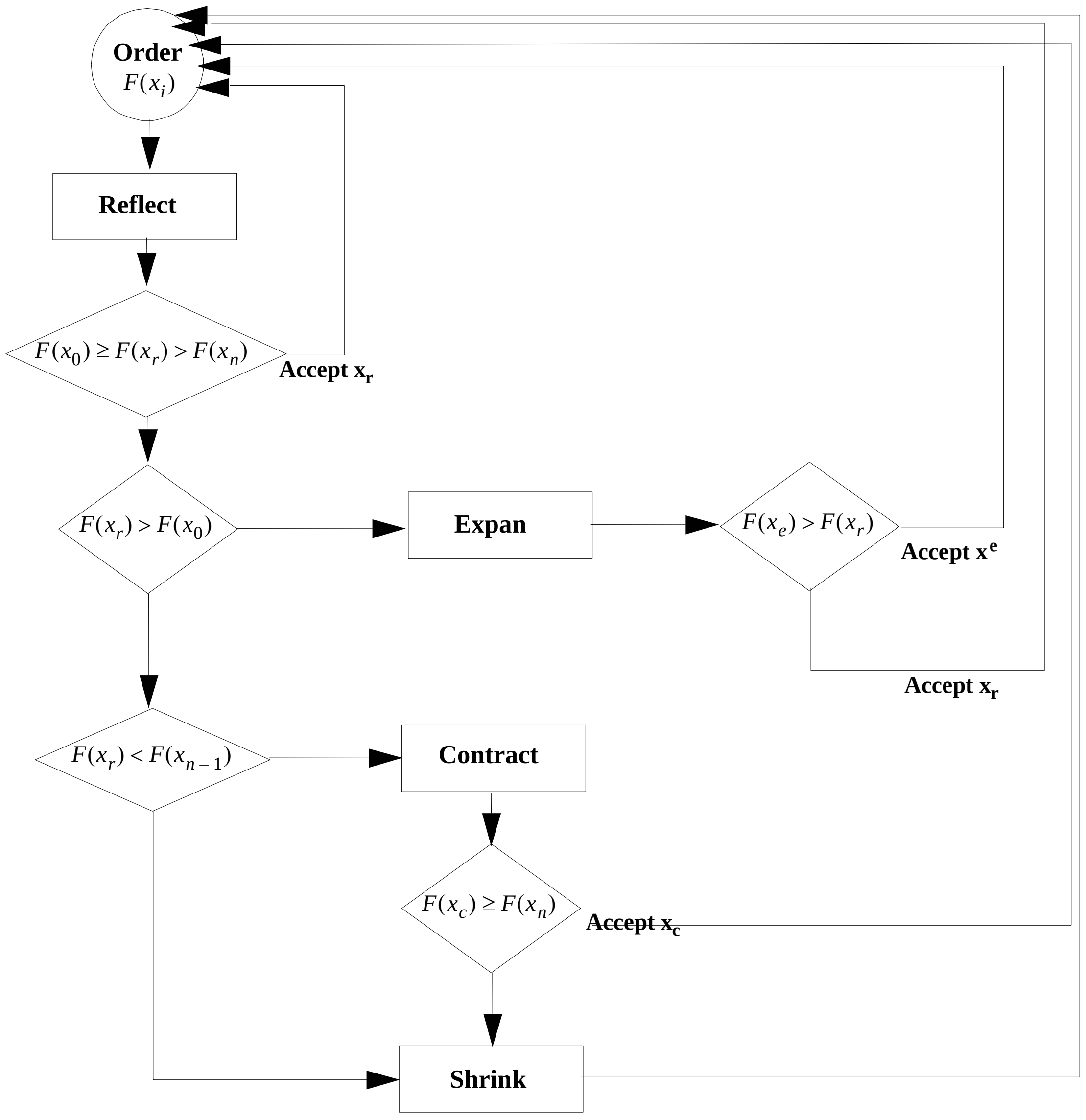
**Figure 117 Amoeba Shrinking**



All vertices move towards best point to form a new amoeba, $x_i = x_0 + \frac{1}{2}(x_i - x_0)$. Note that $x_0$ stays put. It's the best point. Everybody wants to crowd around the best point.

Now that the four movements have been introduced, the algorithm can be explained.

**Figure 118 Amoeba Algorithm Flow**

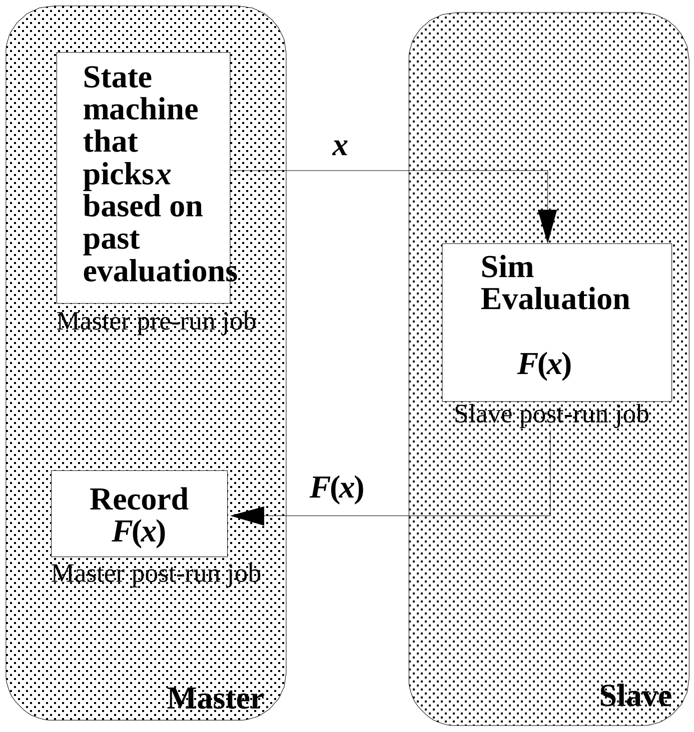


The diamonds in the flowchart are conditionals. The lines coming from the bottom of the diamonds mean the condition proved false. The amoeba tries moving its worst point to a better position. The first shot it takes is moving its badly positioned arm to the reflection point. It evaluates the reflection point, $F(x_r)$. If the evaluation is in between the worst and best it figures this is good enough and stops. If the evaluation at the reflection point is better than the best point, it tries an even bolder move with an expansion. If the reflection point is worse than the second to worst point, it retreats its movement by contracting. If the contraction proves worthy it accepts the contraction point and stops. If all else fails it conservatively moves all legs towards the best point.

## 12.2 The Amoeba In Trick

How is it possible to plug this optimization method into Trick's framework? *F*(*x*), after all, is the actual simulation run. The algorithm relies on evaluating *x* at various stages. The algorithm makes decisions on the fly depending on evaluations. A straightforward approach might take $x_o$,$x_1$ ... $x_n$ and then calculate all of *x*, $x_r$, $x_e$, $x_c$. The simulation could then be ran *n+1+4* times to obtain $F(x_o)$,$F(x_1)$...$F(x_n)$, $F(x)$,$F(x_r)$,$F(x_e)$,$F(x_c)$. With all evaluations in hand, the algorithm could then be marched through. The problem with this is that there would be *n+1+4* simulation run for each iteration. *F*(*x*) is really expensive! In some cases, *F*(*x*) takes minutes! *F*(*x*) needs to be evaluated as little as necessary. After all, *F*(*x*), is an entire simulation run over our input *x*. To minimize the calls to *F*(*x*), it is necessary to build a state machine on one side of the fence.

**Figure 119 State Machine And Sim Evaluations - *F*(*x*)**



## 12.3 Source Code

All the source code for the internals of the amoeba will not be found here. For now, before this algorithm is internalized, you will find amoeba.h and amoeba.c in $TRICK_HOME/trick_models/cannon/optim.

This section contains the source code for the master/slave pre/post run jobs. The majority of the code needed to drive the amoeba is in the state machine for picking *x*. The state machine is found in cannon_pre_master.c. The post-run jobs exist simply to transfer *F*(*x*) from slave to master.

**Figure 120 Slave Passing *F*(*x*) - cannon_post_slave.c**

```
/********************************** TRICK HEADER ***********************
PURPOSE:          (Pass slave sim's evaluation of x to master)
**********************************************************************/
#include "cannon/aero/include/cannon_aero.h"
#include "sim_services/MonteCarlo/include/montecarlo_c_intf.h"
#include "trick_utils/comm/include/tc_proto.h"

int cannon_post_slave( CANNON_AERO* C)`
{
    /* Send F(x) - which is in CANNON_AERO */
    tc_write( mc_get_connection_device(), (char*) C, sizeof(CANNON_AERO) );
    return(0) ;
}
```

**Figure 121 State Machine For Picking *x* - The Heart Of cannon_pre_master.c**

```
while ( 1 ) {
   switch ( A->state ) {
      case VERTICES:
         A->curr_point = A->vertices[A->curr_vertex] ;
         if ( A->curr_vertex == A->num_vertices ) {
            A->state = CALC_CENTROID_POINT ;
         } else {
            fprintf(stderr, "V[%d] ", A->curr_vertex);
         }
         A->curr_vertex++ ;
         break ;
      case CALC_CENTROID_POINT:
         fprintf(stderr, "CENT ");
         amoeba_order( A ) ;
         A->curr_point = A->x_cent ;
         A->state = CALC_REFLECTION_POINT ;
         break ;
      case CALC_REFLECTION_POINT:
         fprintf(stderr, "REFL ");
         amoeba_calc_reflection_point( A ) ;
         A->curr_point = A->x_refl ;
         A->state = REFLECT ;
         break ;
      case REFLECT:
         if ( amoeba_reflect( A ) ) {
            A->state = CALC_CENTROID_POINT ;
         } else {
            fprintf(stderr, "EXPA ");
            amoeba_calc_expansion_point( A ) ;
            A->curr_point = A->x_expa ;
            A->state = EXPAND ;
         }
         break ;
      case EXPAND:
         if ( amoeba_expand( A ) ) {
            A->state = CALC_CENTROID_POINT ;
         } else {
            fprintf(stderr, "CONT ");
            amoeba_calc_contraction_point( A ) ;
            A->curr_point = A->x_cont ;
            A->state = CONTRACT ;
         }
         break ;
      case CONTRACT:
         if ( amoeba_contract( A ) ) {
            A->state = CALC_CENTROID_POINT ;
         } else {
            A->state = SHRINK ;
         }
         break ;
      case SHRINK:
         amoeba_shrink( A ) ;
         fprintf(stderr, "V[0] ");
         A->curr_point = A->vertices[0] ;
         A->curr_vertex = 1 ;
         A->state = VERTICES ;
         break ;
   }
   if ( amoeba_satisfied( A ) && A->state != VERTICES ) {
      exec_terminate( "cannon_pre_master", "Amoeba has found a solution." ) ;
   }
   if ( A->state == CALC_CENTROID_POINT || A->state == SHRINK ) {
      continue ;
   } else {
      break ;
   }
}
C->time_to_fire_jet_1 = A->curr_point[0] ;
C->time_to_fire_jet_2 = A->curr_point[1] ;
C->time_to_fire_jet_3 = A->curr_point[2] ;
C->time_to_fire_jet_4 = A->curr_point[3] ;
```

Let x be an amoeba vertex
$x = x_i$

Let x be the centroid
x = x

Let x be the reflection point
$x = x_r$

Either finished or need to Evaluate expansion point.
Maybe: $x = x_0$

Either finished or need to Evaluate contraction point.
Maybe: $x = x_c$

Either finished or need to shrink the amoeba. X not chosen

Shrink. Get new set $x_0 \ldots x_n$,
$x = x_0$, but will not be evaluated since it remains same.

Current point is the *x* to be evaluated. Here, it is loaded into the proper place for sim evaluation.

**Figure 122 Master Receiving Sim Evaluation - cannon_post_master.c**

```
/****************************** TRICK HEADER *************************
PURPOSE:                 (Read slave sim evaluation)
*******************************************************************/
#include <stdio.h>
#include "cannon/aero/include/cannon_aero.h"
#include "../include/amoeba.h"
#include "sim_services/MonteCarlo/include/montecarlo_c_intf.h"
#include "trick_utils/comm/include/tc_proto.h"

int cannon_post_master(CANNON_AERO* C, AMOEBA* A)
{
       CANNON_AERO C_curr ;
       /* Read slave's results */
       tc_read( mc_get_connection_device(),(char*) &C_curr, sizeof(CANNON_AERO) );

       fprintf(stderr, "%03d> F(", mc_get_current_run());
       amoeba_print_point(4, A->curr_point) ;
       fprintf(stderr, "= %.6lf\n", C_curr.pos[0]);

       /*
        * Load function result for either:
        * simplex vertice, centroid, reflection, contraction or expansion point
        */
       A->curr_point[A->num_dims] = C_curr.pos[0] ;

       return(0) ;
}
```

**Figure 123 Initializing The Amoeba - cannon_init_amoeba.c**

```
/********************** TRICK HEADER***************
PURPOSE:          (Init the Amoeba)
*******************************************************
/
#include <stdio.h>
#include "cannon/aero/include/cannon_aero.h"
#include "../include/amoeba.h"

int cannon_init_amoeba( AMOEBA* A  )
{
       amoeba_init( A, 4, 1.0e-3, 100, NULL, 0.0 ) ;
       return  ( 0 );
}
```

There is a lot of source code there. Note also that in each one of the source files that there is a LIBRARY DEPENDENCY. The dependency is on amoeba.o, whose source is found in amoeba.c. The library dependency tells Trick's CP processor that it will need to compile and link in code that is not explicitly stated in the S_define.

The source code changes for the S_define are below. Instead of adding to the existing "dyn" sim object, a new sim object was added to house the amoeba related functionality. This keeps the "dyn" sim object pure. The choice to create a new sim object is subjective. Sim objects, in theory, should contain related functionality that can be swapped in and out of other S_defines. Note how the dyn's CANNON_AERO* C is accessed out of the "dyn" sim object. A pointer to the CANNON_AERO structure is defined in the OptimSimObject. And at the bottom of the S_define a create_connections function is created to assign the CANNON_AERO pointer to the baseball variable. So when baseball needs to be passed to a job in the optimizer sim object the CANNON_AERO pointer is used (dyn_ptr for our example).

**Figure 124 A New Sim Object - S_define**

```
LIBRARY DEPENDENCIES:
    (
        .
        .
     (cannon/aero/src/cannon_integ_aero.c)
     (cannon/optim/src/cannon_init_amoeba.c)
     (cannon/optim/src/cannon_post_master.c)
     (cannon/optim/src/cannon_pre_master.c)
     (cannon/optim/src/cannon_post_slave.c)
     (cannon/optim/src/amoeba.c)
    )
************************************************************/

#include "sim_objects/monte_trick_sys.sm"

##include "cannon/aero/include/cannon_aero.h"
##include "cannon/aero/include/cannon_monte_proto.h"
##include "cannon/optim/include/amoeba.h"
##include "cannon/optim/include/amoeba_proto.h"

class CannonSimObject : public Trick::SimObject {
        .
        .
};
// Instantiation
CannonSimObject dyn;

class OptimSimObject : public Trick::SimObject {
   public:
       AMOEBA      amoeba ;
       CANNON_AERO *dyn_ptr ;

       OptimSimObject() {
            ("monte_master_init") cannon_init_amoeba( &amoeba ) ;
            ("monte_master_pre")  cannon_pre_master( dyn_ptr, &amoeba ) ;
            ("monte_master_post") cannon_post_master( dyn_ptr, &amoeba ) ;
            ("monte_slave_post")  cannon_post_slave( dyn_ptr ) ;
       }
} ;

// Instantiation
OptimSimObject optimizer ;

collect dyn.baseball.force_collect = {
       dyn.baseball.force_gravity[0],
       dyn.baseball.force_drag[0],
       dyn.baseball.force_magnus[0],
       dyn.baseball.force_cross[0],
       dyn.baseball.force_jet[0] } ;

IntegLoop dyn_integloop (0.01) dyn;

void create_connections() {
   optimizer.dyn_ptr = &dyn.baseball;
}
```

The input file basically remains the same as the Monte Carlo input file except for the inclusion of "amoeba.d" and MonteVarCalculated is used to assign the var#s. The verbosity flag is set to 0 to keep sim messages from cluttering the amoeba printouts.

**Figure 125 Including amoeba.py - input**

```
execfile("Modified_data/cannon_aero.dr")
execfile("Modified_data/amoeba.py")

trick.mc_set_enabled(1)
trick.mc_set_num_runs(70)

trick.mc_set_verbosity(0)
var0 = trick.MonteVarCalculated("dyn.baseball.time_to_fire_jet_1")
var1 = trick.MonteVarCalculated("dyn.baseball.time_to_fire_jet_2")
var2 = trick.MonteVarCalculated("dyn.baseball.time_to_fire_jet_3")
var3 = trick.MonteVarCalculated("dyn.baseball.time_to_fire_jet_4")

trick_mc.mc.add_variable(var0)
trick_mc.mc.add_variable(var1)
trick_mc.mc.add_variable(var2)
trick_mc.mc.add_variable(var3)

dyn.baseball.pos[0] = 0.0
dyn.baseball.pos[1] = 0.0
dyn.baseball.pos[2] = 0.0

dyn.baseball.vel[0] = 43.30
dyn.baseball.vel[1] = 0.0
dyn.baseball.vel[2] = 25.0

dyn.baseball.theta  = trick.attach_units("d" , -90.0)
dyn.baseball.phi    = trick.attach_units("d" , 1.0)
dyn.baseball.omega0 = trick.attach_units("rev/s" , 30.0)

dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 6)

trick.stop(10.0)
```

The amoeba default data file is used to initialize the AMOEBA data structure. Its main purpose is to setup a first-try amoeba i.e., the first guess. Each row of the init_simplex matrix is a set of jet firing times followed by the resulting distance the cannon flies given the jet firing sequence.

**Figure 126 Default Data For Amoeba - amoeba.py**

```
optimizer.amoeba.debug = 0

#Allocate a 5x5 matrix that holds simplex and function results
optimizer.amoeba.init_simplex = trick.alloc_type(5, "double *")
optimizer.amoeba.init_simplex[0] = trick.alloc_type(5, "double")
optimizer.amoeba.init_simplex[1] = trick.alloc_type(5, "double")
optimizer.amoeba.init_simplex[2] = trick.alloc_type(5, "double")
optimizer.amoeba.init_simplex[3] = trick.alloc_type(5, "double")
optimizer.amoeba.init_simplex[4] = trick.alloc_type(5, "double")

#Note that function results are in the 5th column.  They will be
#calculated by the simulation.  For now we set them to zero.
optimizer.amoeba.init_simplex[0] =  [4.0, 4.5, 5.0, 6.0, 0.0]
optimizer.amoeba.init_simplex[1] =  [3.5, 4.0, 4.5, 5.0, 0.0]
optimizer.amoeba.init_simplex[2] =  [4.5, 5.0, 5.5, 6.5, 0.0]
optimizer.amoeba.init_simplex[3] =  [3.0, 3.5, 4.0, 4.5, 0.0]
optimizer.amoeba.init_simplex[4] =  [5.0, 5.5, 6.5, 7.0, 0.0]
```

## 12.4 Building The Simulation

No pain, no gain. In this section, we'll revise that to "No pain, no pain." We will cheat a little. There is just too much code! We'll copy the source out of $TRICK_HOME where it is nicely stored for your consumption.

```
UNIX Prompt> cd $HOME/trick_models/cannon
UNIX Prompt> cp -r $TRICK_HOME/trick_models/cannon/optim  optim
UNIX Prompt> cp $TRICK_HOME/trick_models/cannon/aero/src/cannon_impact_monte.c
             aero/src
UNIX Prompt> cp $TRICK_HOME/trick_models/cannon/aero/include/cannon_monte_proto.h
             aero/include
UNIX Prompt> cd $HOME/trick_sims
UNIX Prompt> cp -r $TRICK_HOME/trick_sims/SIM_amoeba SIM_cannon_amoeba
UNIX Prompt> cd SIM_cannon_amoeba
UNIX Prompt> CP
UNIX Prompt> ./S_main*exe RUN_test/input.py
```

## 12.5 Results

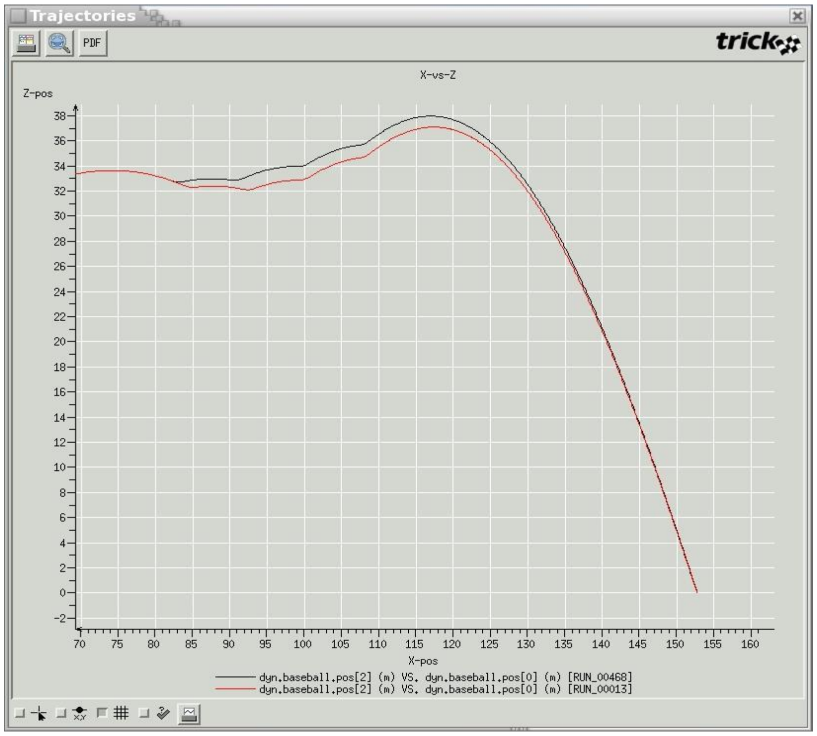
The amoeba fairs well. In 66 simulation runs (or function evaluations) it crawls its way on a very jagged ugly path to a solution. What is funny is that the amoeba actually finds its best sequence *x*=(3.4, 3.9, 4.4, 5.0) on run 13 while evaluating the centroid point. This point -really- didn't need to be evaluated as it is not part of the algorithm's core. In fact, the amoeba never takes this best point into consideration. A missing gold mine. Oh well. We'll keep the centroid evaluation in case it helps anybody else.

The amoeba's solution outdoes the Gaussian attempt.

```
Amoeba: x=(3.4, 3.9, 4.4, 5.0) Yielding a distance => 152.790802m
Gaussi: x=(3.3, 3.8, 4.4, 5.0) Yielding a distance => 152.790795m
```

The amoeba inches out by 7 micrometers. The plot below shows a comparison between running with the jet firing times from the Amoeba solution versus the Gaussian solution. Interestingly, the curves are distinctly different but converge on impact. Note that the amoeba's curve trails underneath the Gaussian curve almost until impact.

**Figure 127 Amoeba Versus Gauss**



It took the Gaussian method 468 random attempts before it arrived at its solution, compared to the amoeba's 66 --- well really 13 tries. Hmmm? Shall we abandon the purely random method??? Remember that the amoeba's path is dependent on the initial simplex. We tried several other initial simplexes where the amoeba crawled into the wrong hole. It's not fail safe by any stretch.

## 12.6 Future

There is much to be learned on our side, and much to implement in the way of optimization. The amoeba algorithm scratches the surface of an entire field. Note that the amoeba algorithm itself was fairly removed from the simulation. Soon this algorithm, and hopefully a host of others, will be buried beneath the scenes with a simple interface for optimizing simulations generically. If you have any ideas, or algorithms to share, please post it on the Trick bulletin board (http://trick.jsc.nasa.gov).

## 13.0 Conclusion

Hopefully you found the tutorial beneficial. We are very interested in any comments you have. Post them to the Trick bulletin board (http://trick.jsc.nasa.gov). Looking back, we hope that you learned:

- How to set yourself up with the Trick environment.
- Why to use Trick in the first place rather than a home-baked simulation.
- How to integrate source code into a Trick simulation executable using the simulation definition file (S_define).
- How to use Trick’s data products for viewing/comparing data.
- How to view real-time data with stripcharts and "Trick View" (TV).
- How and why to use Trick’s derivative and integration routines.
- How and why to use dynamic events for contact.
- How to use Trick’s socket communication package.
- How and why to use the collection mechanism for force summations.
- How to take advantage of Trick’s input processor.
- How to write a GUI to connect to Trick’s "variable server" to drive the simulation.
- How to run Monte Carlo simulations
- How to plug into Trick’s optimization framework (the amoeba example).

We hope that you aren’t walking away:

- Lost
- Frustrated
- Confused
- Sick
- Mad!
- All of the above

THE END.