Tutorial Review

Trick 17

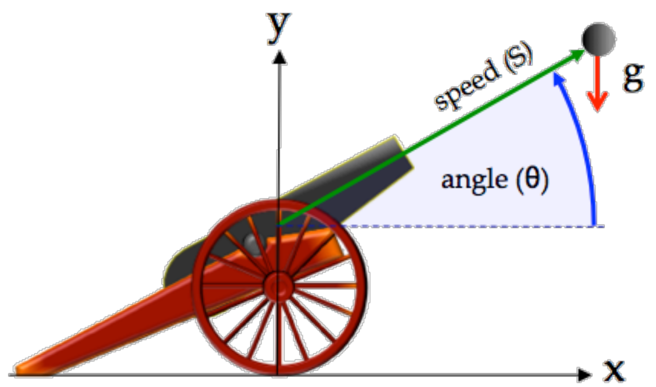
by The Trick Team

This presentation is a review of the Trick Tutorial, which can be found at the following URL:

https://github.com/nasa/trick/wiki/Tutorial

Example Dynamics Problem

Determine the trajectory, and time of impact of a cannon ball that is fired with an initial speed and initial angle. Assume a constant acceleration of gravity (g), and no aerodynamic forces.



Analytic Solution for Cannon Ball

Acceleration is the second-derivative of position with respect to time.

$$\frac{d^2\vec{p}}{dt^2} = \vec{a}(t)$$

If *a(t)* is integratable, then we can find an analytic solution. In the case of our cannon ball problem, *a(t)=g* is constant, so our solution is:

$$\vec{v}(t) = \vec{a}t + \vec{v}_0$$ (velocity- anti-derivative of acceleration)

$$\vec{p}(t) = \frac{1}{2}\vec{a}t^2 + \vec{v}_0 t + \vec{p}_0$$ (position- anti-derivative of velocity)

Initial Conditions for Cannon Ball

$S = 50$ Speed of the cannon ball

$\theta = \pi / 6$ (30°) Angle of the barrel

$g = -9.81$ Acceleration of gravity

$$\vec{a} = \begin{bmatrix} a_x \\ a_y \end{bmatrix} = \begin{bmatrix} 0 \\ g \end{bmatrix}$$

Acceleration

$$\vec{v}_0 = \begin{bmatrix} v_{0x} \\ v_{0y} \end{bmatrix} = \begin{bmatrix} S \cdot \cos\theta \\ S \cdot \sin\theta \end{bmatrix}$$

The initial velocity of the cannon ball

$$\vec{p}_0 = \begin{bmatrix} p_{0x} \\ p_{0y} \end{bmatrix} = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}$$

The initial position of the cannon ball

Cannon Ball Time of Impact

The Ball hits the ground when:

$$p_y(t) = \frac{1}{2} a_y t^2 + v_{0y} t + p_{0y} = 0$$

Solving for t, using the quadratic formula:

$$t_{impact} = \frac{-v_{0y} - \sqrt{v_{0y}^2 - 2p_{0y}}}{a_y}$$

A Simple (non-Trick) Simulation

Cannon.c (page 1 of 2)

```c
/* Cannonball without Trick */
#include <stdio.h>
#include <math.h>

int main (int argc, char * argv[]) {
    /* Declare variables used in the 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 ;
    int impact;
    double impactTime;
    /* 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 ;
    impact = 0;
```

Cannon.c (page 2 of 2)

```c
 /* 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 */
    printf("time, pos[0], pos[1], vel[0], vel[1]\n" );
    while ( !impact ) {
        vel[0] = vel_orig[0] + acc[0] * time ;
        vel[1] = vel_orig[1] + acc[1] * time ;
        pos[0] = pos_orig[0] + (vel_orig[0] + 0.5 * acc[0] * time) * time ;
        pos[1] = pos_orig[1] + (vel_orig[1] + 0.5 * acc[1] * time) * time ;
        printf("%7.2f, %10.6f, %10.6f, %10.6f, %10.6f\n",
            time, pos[0], pos[1], vel[0], vel[1] );
        if (pos[1] < 0.0) {
            impact = 1;
            impactTime = (- vel_orig[1] -
                          sqrt(vel_orig[1] * vel_orig[1] - 2.0 * pos_orig[1])
                         ) / -9.81;
            pos[0] = impactTime * vel_orig[0];
            pos[1] = 0.0;
        }
        time += 0.01 ;
    }
    /* Shutdown simulation */
        printf("Impact time=%lf position=%lf\n", impactTime, pos[0]);
    return 0;
}
```

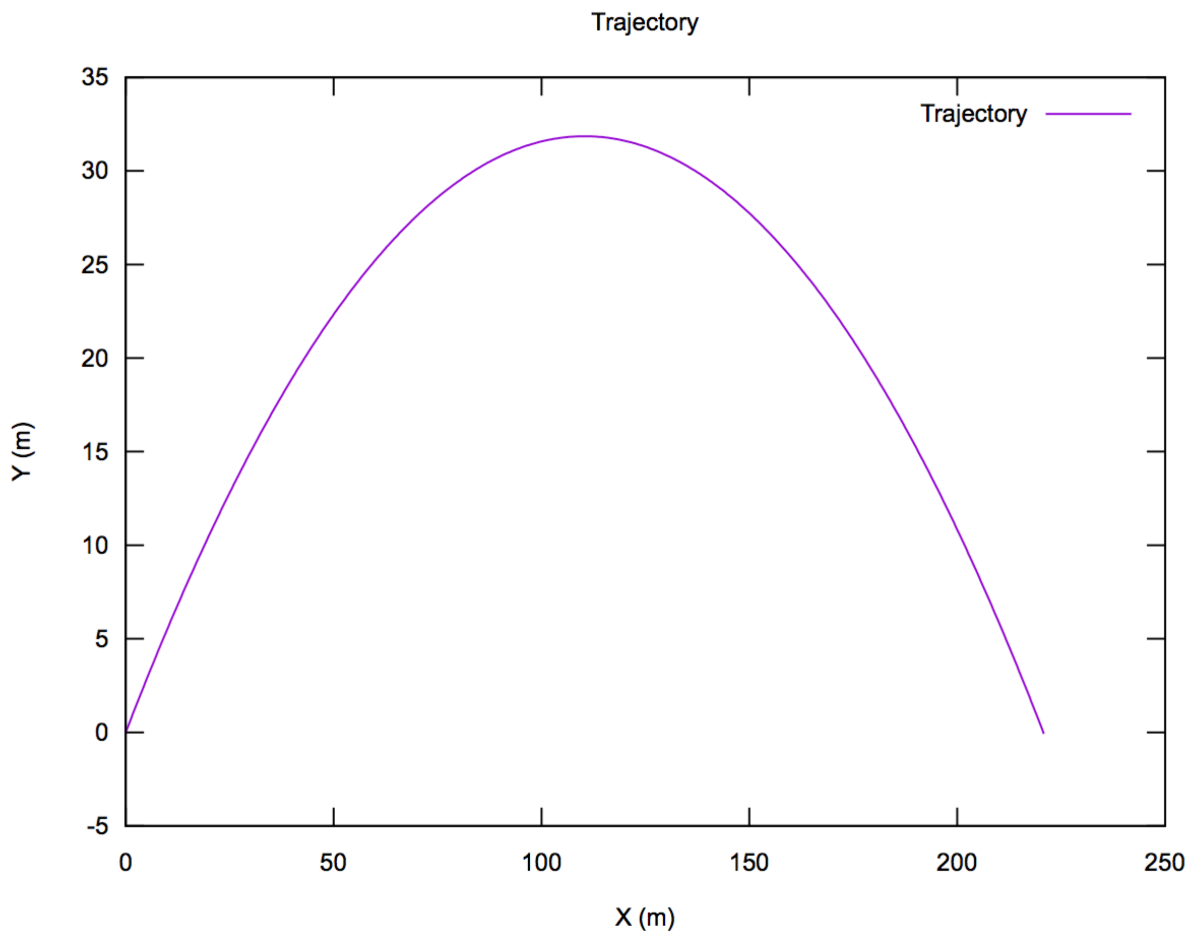
If we compile and run the program :

```
% cc cannon.c -o cannon -lm
% ./cannon
```

We will get a listing of the trajectory points over time, followed by the impact position and time:

```
Impact time=5.096840 position=220.699644
```

If we plot the trajectory points using Gnuplot:



So why do we need Trick?

Limitations of the Simulation

For simple physics models like our cannonball, maybe we don't need Trick, but real-world problems are rarely as simple.

1. Real-world problems don't often have nice closed-form solutions. They require numeric methods.

2. Changing parameters requires recompilation of the program.

3. What if we want to be able to run our simulation in real-time?

4. What if we want to interact with our simulation while its running?

In the coming sections, we'll see how Trick helps us overcome these limitations, and more. We'll see how it provides simulations with commonly needed capabilites, many of them **automatically**.

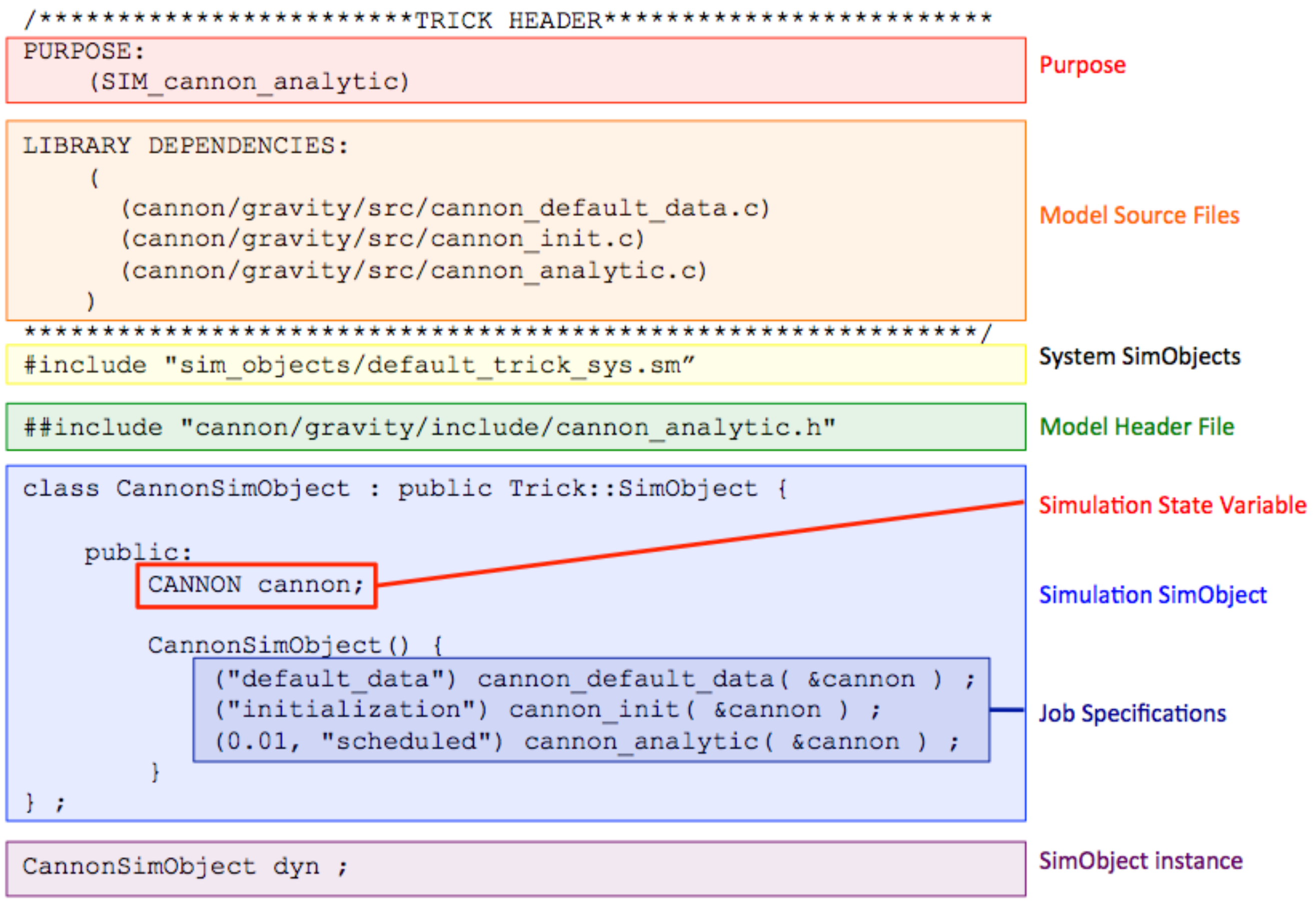
The Simulation Definition File (S_define)

In our non-Trick simulation program :

- We created a representation of our simulation-state.
- We defined our simulation-state variables.
- We organized our code into well-defined tasks :
    - Initialization of variables by assignment
    - Initialization of variables by calculation.
    - Periodic calculation of the model state.
    - Cleanup/Shutdown.
- We made sure that our tasks were executed in the appropriate order.

In a Trick simulation, we do the same. To "teach" Trick about the parts that make the particular simulation unique, we use a simulation definition file (**S_define**).

Example S_define

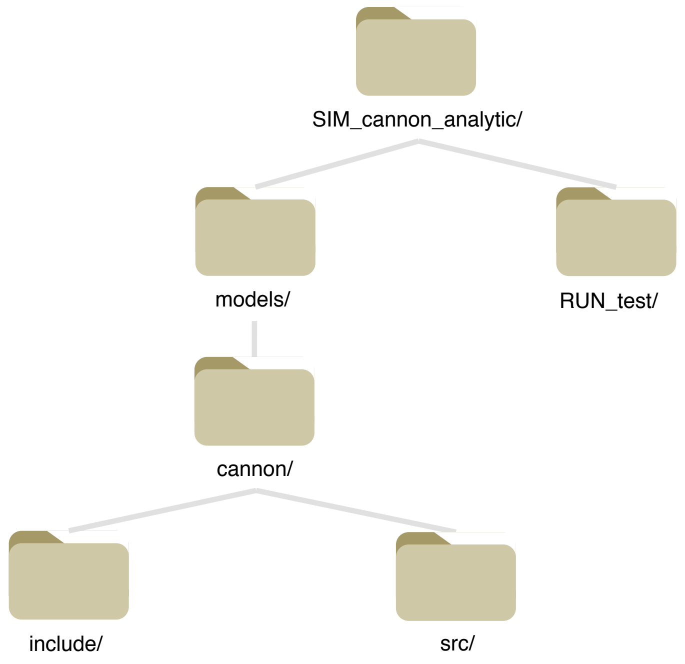


```
/************************TRICK HEADER*************************
PURPOSE:
    (SIM_cannon_analytic)

LIBRARY DEPENDENCIES:
    (
      (cannon/gravity/src/cannon_default_data.c)
      (cannon/gravity/src/cannon_init.c)
      (cannon/gravity/src/cannon_analytic.c)
    )
*************************************************************/
#include "sim_objects/default_trick_sys.sm”

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

class CannonSimObject : public Trick::SimObject {

    public:
        CANNON cannon;

        CannonSimObject() {
            ("default_data") cannon_default_data( &cannon ) ;
            ("initialization") cannon_init( &cannon ) ;
            (0.01, "scheduled") cannon_analytic( &cannon ) ;
        }
} ;

CannonSimObject dyn ;
```

Building & Running a Trick Simulation

Simulation File Organization

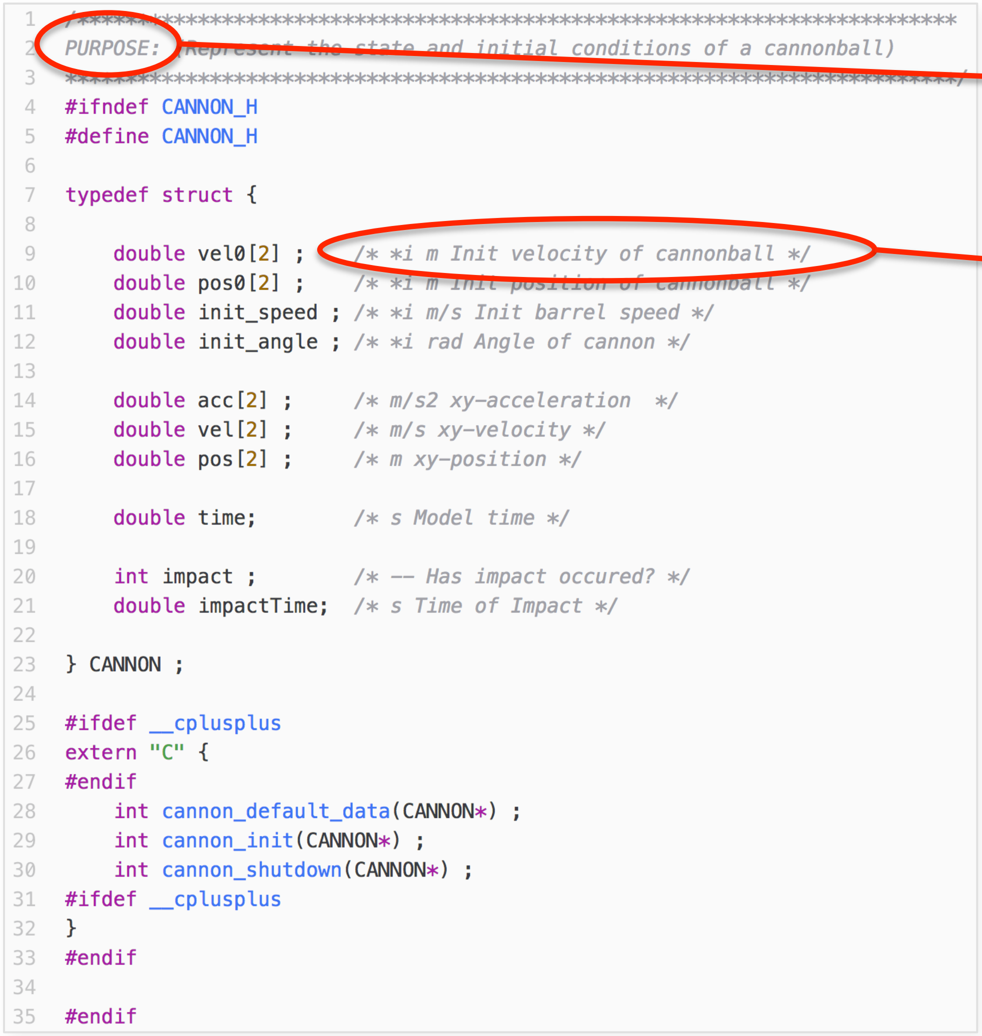
```
% mkdir -p SIM_cannon_analytic/RUN_test
% mkdir -p SIM_cannon_analytic/models/cannon/src
% mkdir -p SIM_cannon_analytic/models/cannon/include
```



trick

Representing the Cannonball State

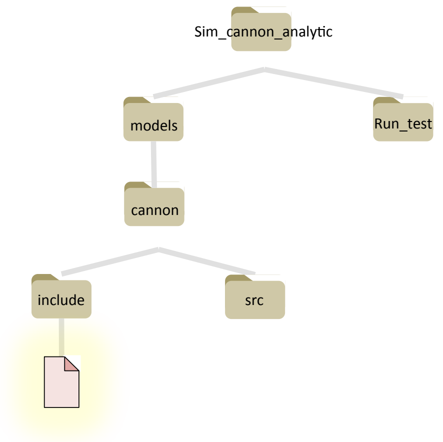
cannon.h



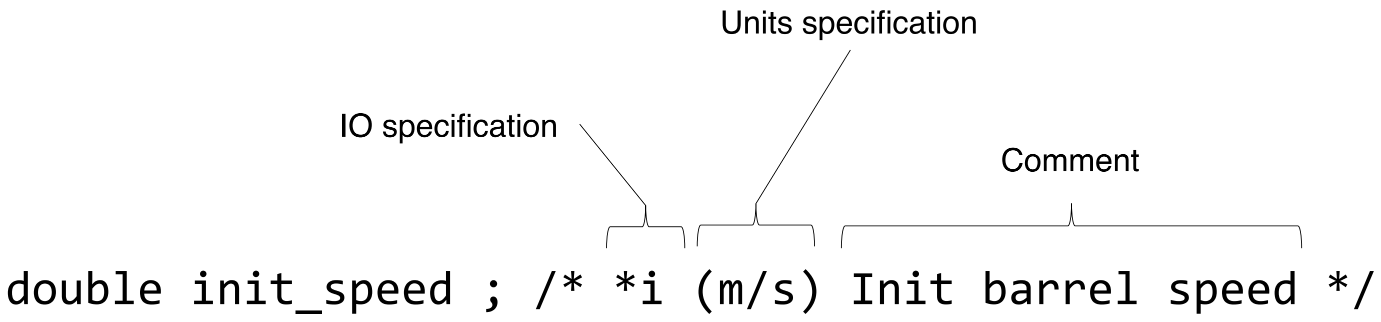
```
/*************************************************************************
PURPOSE: (Represent the state and initial conditions of a cannonball)
**************************************************************************/
#ifndef CANNON_H
#define CANNON_H

typedef struct {

    double vel0[2] ;    /* *i m Init velocity of cannonball */
    double pos0[2] ;    /* *i m Init position of cannonball */
    double init_speed ; /* *i m/s Init barrel speed */
    double init_angle ; /* *i rad Angle of cannon */

    double acc[2] ;     /* m/s2 xy-acceleration  */
    double vel[2] ;     /* m/s xy-velocity */
    double pos[2] ;     /* m xy-position */

    double time;        /* s Model time */

    int impact ;        /* -- Has impact occured? */
    double impactTime;  /* s Time of Impact */

} CANNON ;

#ifdef __cplusplus
extern "C" {
#endif
    int cannon_default_data(CANNON*) ;
    int cannon_init(CANNON*) ;
    int cannon_shutdown(CANNON*) ;
#ifdef __cplusplus
}
#endif

#endif
```

Keyword that tells Trick to process this file.

Specially formatted comments for Trick.



Trick Comments

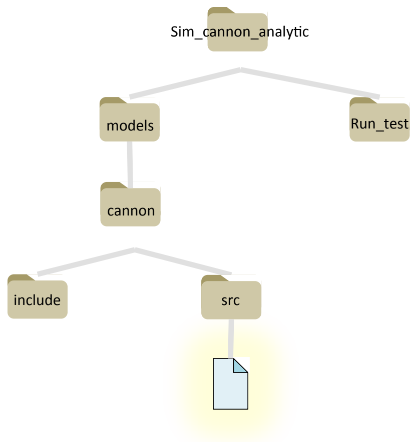


```
double init_speed ; /* *i (m/s) Init barrel speed */
```

Initializing the Cannonball State

cannon_init.c

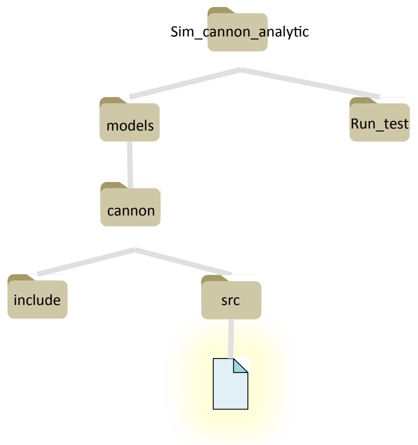
```c
/********************************* TRICK HEADER *****************************
PURPOSE: (Set the initial data values)
***************************************************************************/

/* Model Include files */
#include <math.h>
#include "../include/cannon.h"

/* default data job */
int cannon_default_data( CANNON* C ) {

    C->acc[0] = 0.0;
    C->acc[1] = -9.81;
    C->init_angle = M_PI/6 ;
    C->init_speed  = 50.0 ;
    C->pos0[0] = 0.0 ;
    C->pos0[1] = 0.0 ;

    C->time = 0.0 ;

    C->impact = 0 ;
    C->impactTime = 0.0 ;

    return 0 ;
}

/* initialization job */
int cannon_init( CANNON* C) {

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

    C->vel[0] = C->vel0[0] ;
    C->vel[1] = C->vel0[1] ;

    C->impactTime = 0.0;
    C->impact = 0.0;

    return 0 ;
}
```



Updating the Cannonball State Over Time

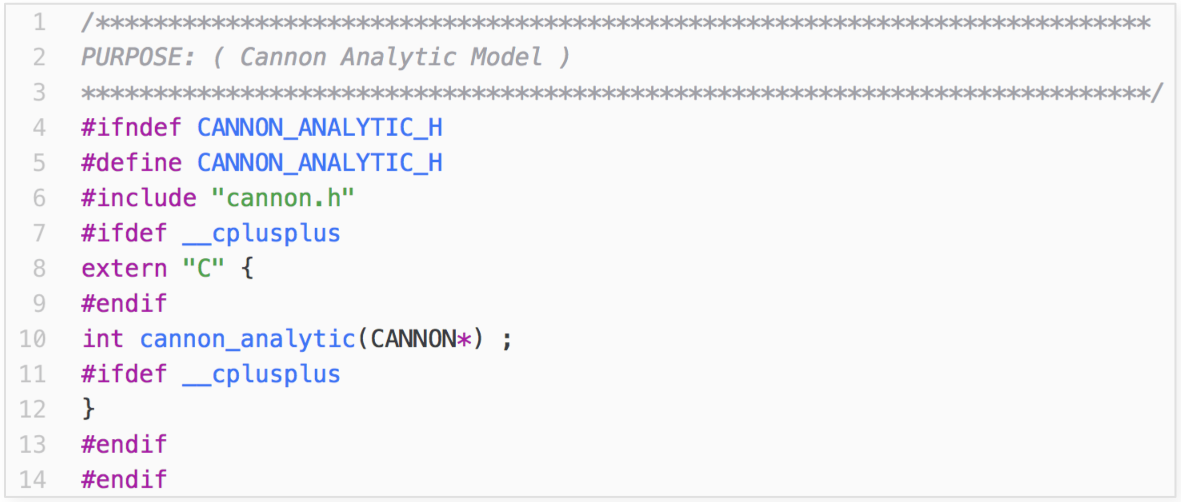
cannon_analytic.c

```c
/*****************************************************************************
PURPOSE:    ( Analytical Cannon )
*****************************************************************************/
#include <stdio.h>
#include <math.h>
#include "../include/cannon_analytic.h"

int cannon_analytic( CANNON* C ) {

    C->acc[0] =  0.00;
    C->acc[1] = -9.81 ;
    C->vel[0] = C->vel0[0] + C->acc[0] * C->time ;
    C->vel[1] = C->vel0[1] + C->acc[1] * C->time ;
    C->pos[0] = C->pos0[0] + (C->vel0[0] + (0.5) * C->acc[0] * C->time) * C->time ;
    C->pos[1] = C->pos0[1] + (C->vel0[1] + (0.5) * C->acc[1] * C->time) * C->time ;
    if (C->pos[1] < 0.0) {
        C->impactTime = (- C->vel0[1] - sqrt( C->vel0[1] * C->vel0[1] - 2 * C->pos0[1]))/C->acc[1];
        C->pos[0] = C->impactTime * C->vel0[0];
        C->pos[1] = 0.0;
        C->vel[0] = 0.0;
        C->vel[1] = 0.0;
        if ( !C->impact ) {
            C->impact = 1;
            fprintf(stderr, "\n\nIMPACT: t = %.9f, pos[0] = %.9f\n\n", C->impactTime, C->pos[0] ) ;
        }
    }
    /*
     * 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.
     */
    C->time += 0.01 ;
    return 0 ;
}
```

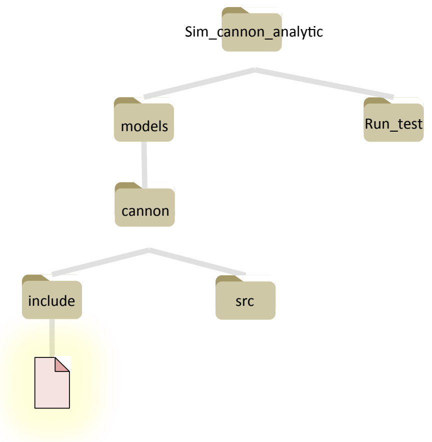


We Need a Prototype for Our State Update Approach

cannon_analytic.h

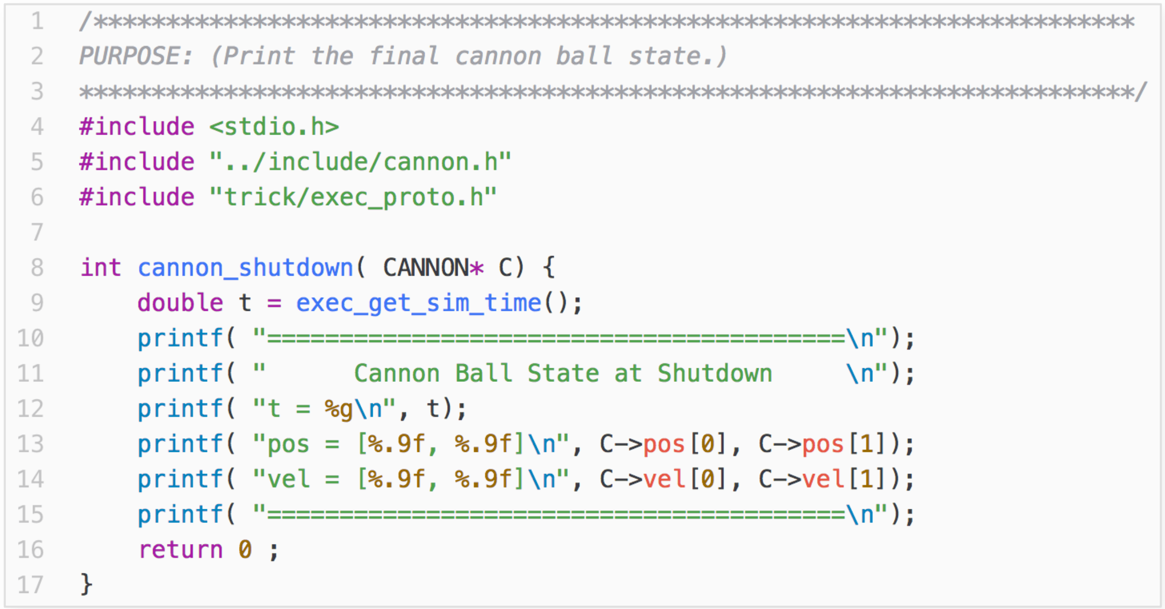


```
/*************************************************************************
PURPOSE: ( Cannon Analytic Model )
**************************************************************************/
#ifndef CANNON_ANALYTIC_H
#define CANNON_ANALYTIC_H
#include "cannon.h"
#ifdef __cplusplus
extern "C" {
#endif
int cannon_analytic(CANNON*) ;
#ifdef __cplusplus
}
#endif
#endif
```

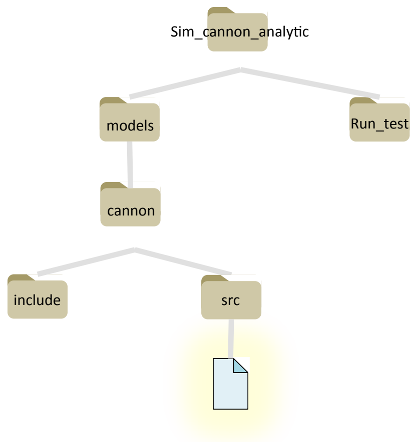


Cannonball Cleanup And Shutdown

cannon_shutdown.c



```c
/*****************************************************************************
PURPOSE: (Print the final cannon ball state.)
*****************************************************************************/
#include <stdio.h>
#include "../include/cannon.h"
#include "trick/exec_proto.h"

int cannon_shutdown( CANNON* C) {
    double t = exec_get_sim_time();
    printf( "========================================\n");
    printf( "      Cannon Ball State at Shutdown     \n");
    printf( "t = %g\n", t);
    printf( "pos = [%.9f, %.9f]\n", C->pos[0], C->pos[1]);
    printf( "vel = [%.9f, %.9f]\n", C->vel[0], C->vel[1]);
    printf( "========================================\n");
    return 0 ;
}
```



The Simulation Definition File

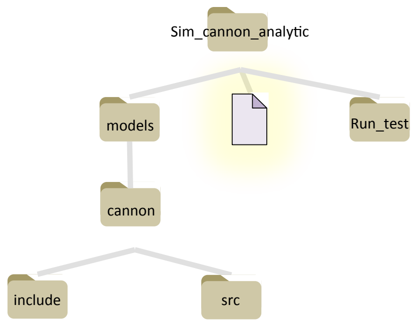
S_define

```
/*************************TRICK HEADER*************************
PURPOSE:
    (This S_define works with the RUN_analytic input file)
LIBRARY DEPENDENCIES:
    (
      (cannon/src/cannon_init.c)
      (cannon/src/cannon_analytic.c)
      (cannon/src/cannon_shutdown.c)
    )
*************************************************************/

#include "sim_objects/default_trick_sys.sm"
##include "cannon/include/cannon_analytic.h"

class CannonSimObject : public Trick::SimObject {

    public:
        CANNON cannon;

        CannonSimObject() {
            ("default_data") cannon_default_data( &cannon ) ;
            ("initialization") cannon_init( &cannon ) ;
            (0.01, "scheduled") cannon_analytic( &cannon ) ;
            ("shutdown") cannon_shutdown( &cannon ) ;
        }
} ;

CannonSimObject dyn ;
```

Trick Header

Included files

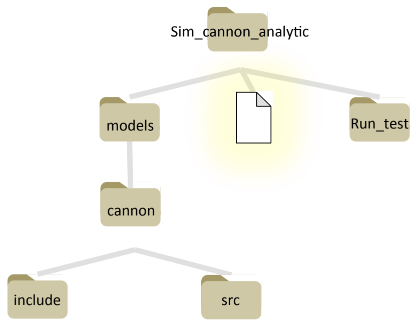
Jobs



Compiler Flags

S_overrides.mk

```
TRICK_CFLAGS += -Imodels
TRICK_CXXFLAGS += -Imodels
```



The Importance of **TRICK_CFLAGS** and **TRICK_CXXFLAGS**

`$TRICK_CFLAGS` and `$TRICK_CXXFLAGS` are Trick environment variables. They provide a means to control how your simulation is built.

When Trick invokes a C-compiler, it passes the value of `$TRICK_CFLAGS` to it. `$TRICK_CXXFLAGS` are passed to C++ compilers.

For example, suppose I want the C compiler to warn me about common, dubious code constructs that might have crept into my simulation code:

```
    TRICK_CFLAGS += -Wall
```

The Trick build process also uses any -I options found in these variables to resolve relative file paths.

The Importance of **`TRICK_CFLAGS`** and **`TRICK_CXXFLAGS`**

The –**I** flag is a GNU compiler flag that lists **base-paths**, from which relative-paths, specified in your code, can be resolved to full-paths.

For example, the following directive specifies a **relative-path**:

```
#include “cannon/gravity/include/cannon.h”
```

It’s relative to some **base-path** that needs to be specified in order to find the file.

So, when we provide the following base-path, the compiler can resolve the relative path.

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

Example:

$$\overbrace{\texttt{\$\{HOME\}/trick\_models/}}^{\text{Base-path}}\overbrace{\texttt{cannon/gravity/include/cannon.h}}^{\text{Relative-path}}$$

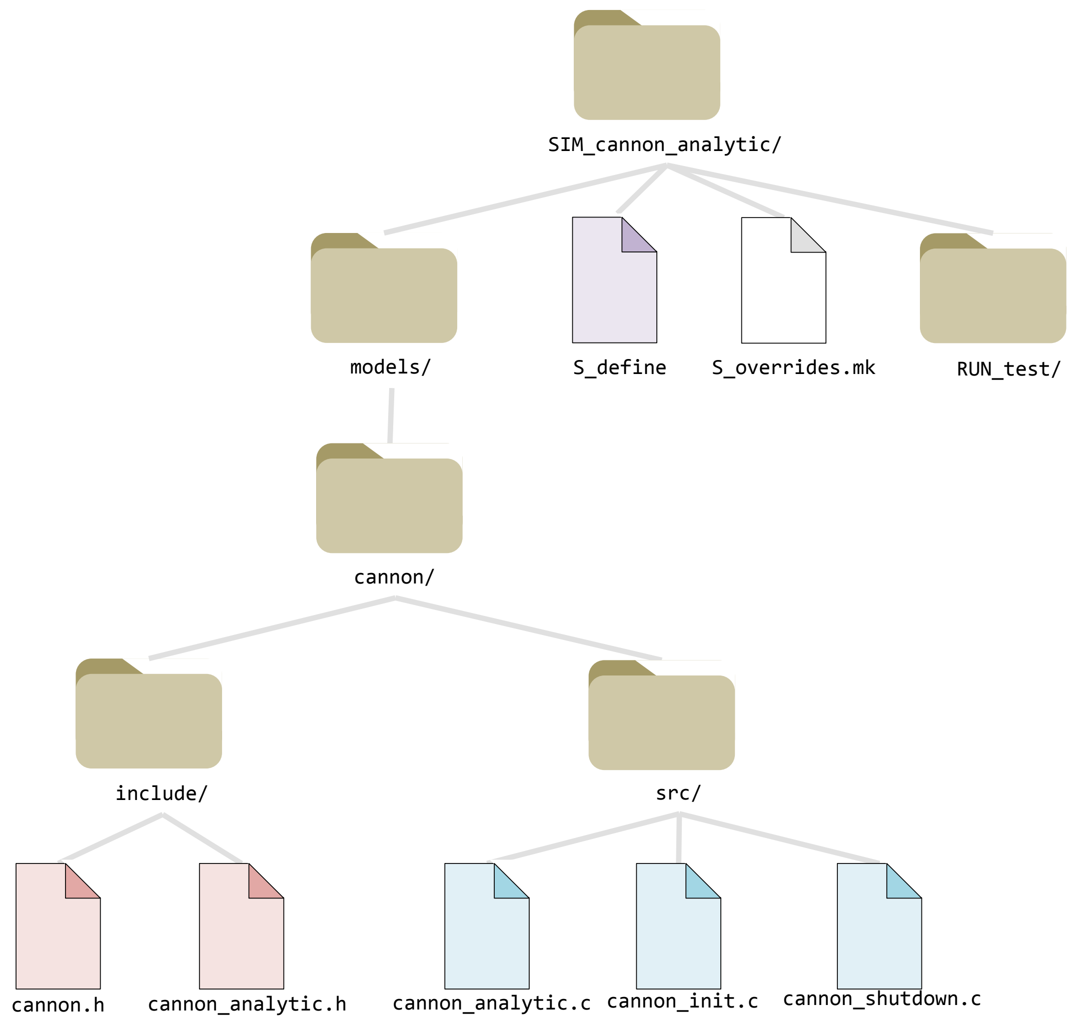
trick

The Importance of **`TRICK_CFLAGS`** and **`TRICK_CXXFLAGS`**

The –I flags in `TRICK_CFLAGS` and `TRICK_CXXFLAGS` are also used to resolve relative paths in `LIBRARY DEPENDENCIES` specifications in Trick headers.

```
LIBRARY DEPENDENCIES:
    (
      (cannon/src/cannon_init.c)
      (cannon/src/cannon_analytic.c)
      (cannon/src/cannon_shutdown.c)
    )
```

Simulation File Organization



Building the Simulation with *Trick-CP*

The Trick simulation build tool is called **trick-CP** (Trick Configuration Processor). It parses an S_define file, finding data-types, variables, functions, scheduling information, and ultimately creates a simulation executable.

After you build the sim:

👉
```
% cd $HOME/trick_sims/SIM_cannon_analytic
% trick-CP
```

you should see:

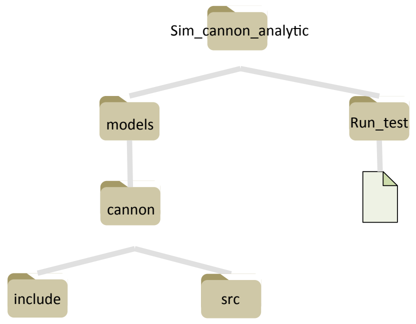
```
=== Simulation make complete ===
```

Simulation Input File

Every Trick simulation needs an input file. The input file is actually a script that is processed by a Python interpreter, specifically bound to Trick's code and your simulation code.

input.py

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



Simulation Input File

Run the simulation executable **from** SIM_cannon_analytic/ :

```
% ./S_main_*.exe RUN_test/input.py
```

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

```
IMPACT: t = 5.096839959, pos[0] = 220.699644186

========================================
      Cannon Ball State at Shutdown
t = 5.2
pos = [220.699644186, 0.000000000]
vel = [0.000000000, 0.000000000]
========================================
REALTIME SHUTDOWN STATS:
     REALTIME TOTAL OVERRUNS:            0
            ACTUAL INIT TIME:        0.099
         ACTUAL ELAPSED TIME:       11.338
SIMULATION TERMINATED IN
  PROCESS: 0
  ROUTINE: Executive_loop_single_thread.cpp:98
  DIAGNOSTIC: Reached termination time

       SIMULATION START TIME:        0.000
        SIMULATION STOP TIME:        5.200
     SIMULATION ELAPSED TIME:        5.200
        ACTUAL CPU TIME USED:        0.098
       SIMULATION / CPU TIME:       53.268
     INITIALIZATION CPU TIME:        0.052
```

Result

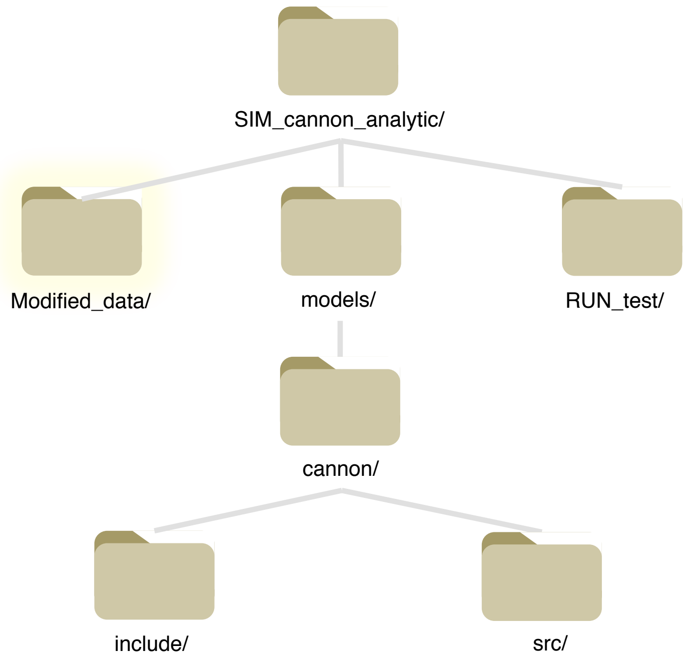
What about the trajectory?

To plot the trajectory, we first need to record the data from our simulation.

Recording Simulation Data

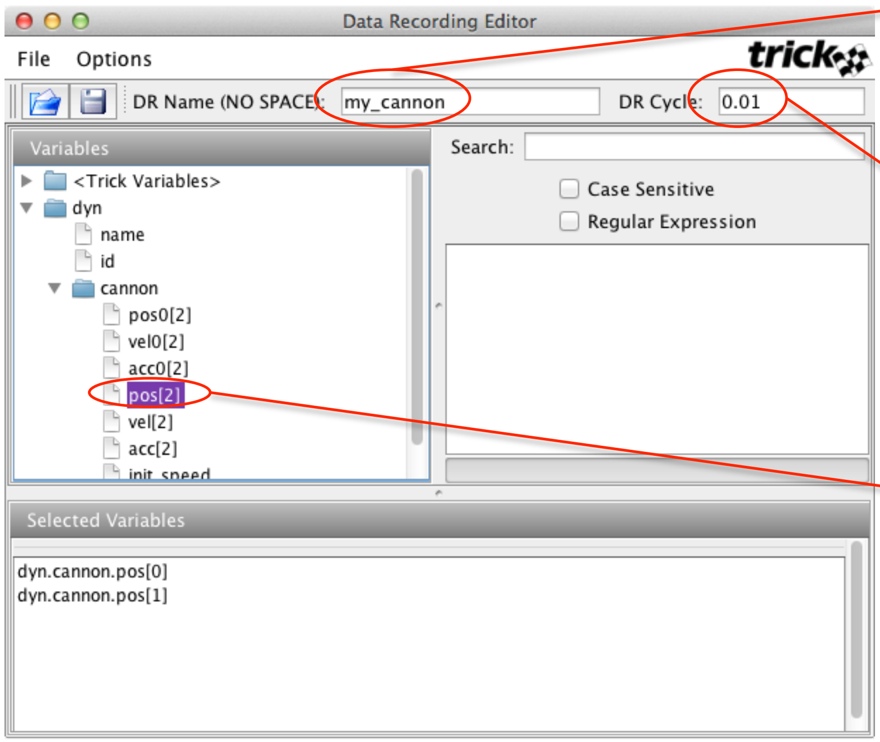
To tell our sim what to record, we need to create a **data-recording file**. Let's first make a place to put it:

👉
```
% mkdir Modified_data
```



Making a Data-Recording File with *trick-dre*

```
% trick-dre &
```



DR Name (NO SPACE): my_cannon
DR Cycle: 0.01

```
dyn.cannon.pos[0]
dyn.cannon.pos[1]
```

1) Give the variable collection (a.k.a "recording group") a name.

2) Specify the recording frequency. Ok, fine, the period.

3) Double-click on the pos array (the position of the cannon ball). Note that they are then dsiplayed in the Selected variables pane.

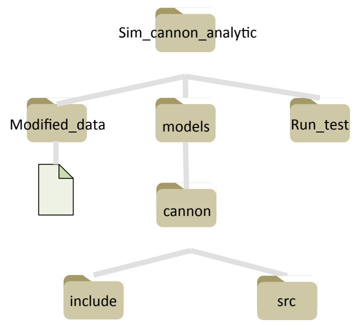
4) Choose File->Save. In the "Save" dialog, enter the file name "cannon.dr" and save it in the Modified_data/ directory.
5) Exit trick-dre.

What is a Data-Recording File?

cannon.dr

```python
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.DRBinary("my_cannon"))
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("dyn.cannon.pos0[0]")
drg[DR_GROUP_ID].add_variable("dyn.cannon.pos0[1]")
trick.add_data_record_group(drg[DR_GROUP_ID], trick.DR_Buffer)
drg[DR_GROUP_ID].enable()
```

DON'T TYPE THIS IN.
IT'S CREATED AUTOMATICALLY
BY trick-dre.



A data recording file is actually a snippet of Python code that you include into your input file.

Initiation of Data Recording

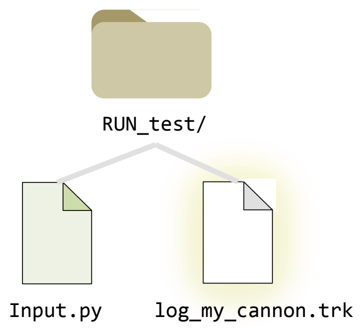
Update input.py

```python
execfile("Modified_data/cannon.dr")
trick.stop(5.2)
```

Re-run the simulation:

```
% ./S_main*.exe RUN_test/input.py
```

This will produce a data recording file in your RUN_ directory.



Trick Data Products

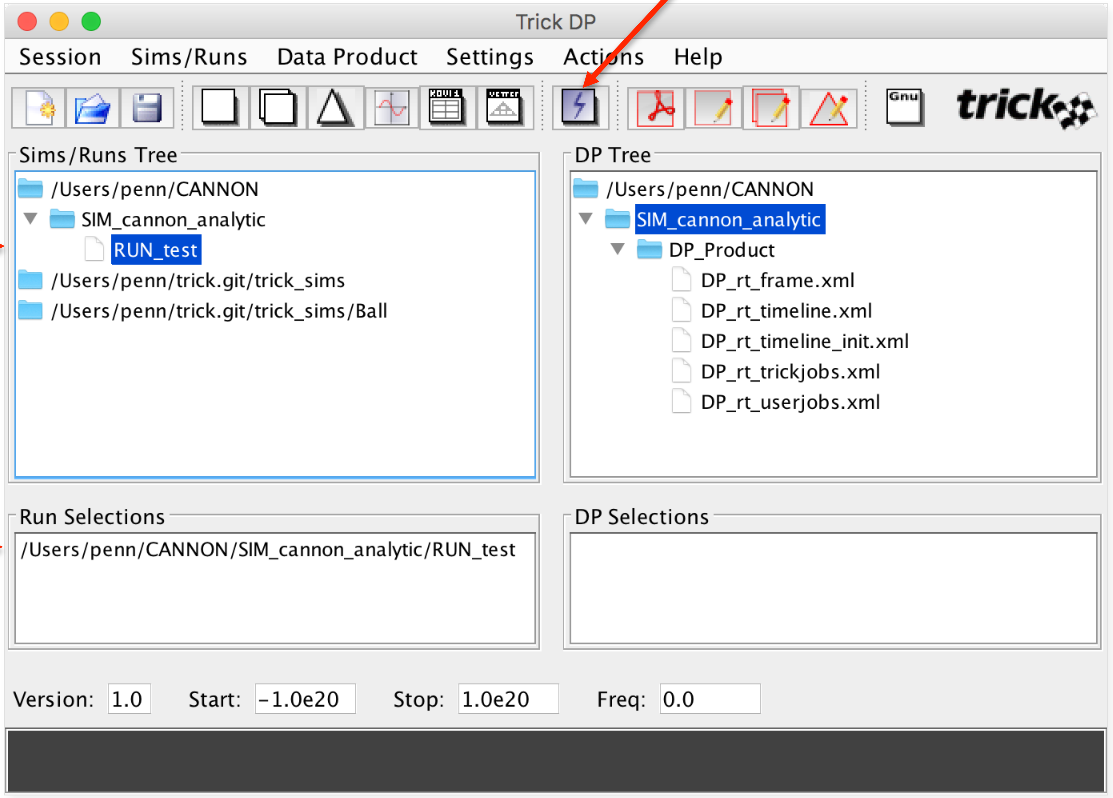
In the SIM_cannon_analytic/ directory, run *trick-dp*:

```
% trick-dp &
```

2) Click to start trick-qp

double-click to select the recorded data.

Selected data will appear here.



Sims/Runs Tree

```
/Users/penn/CANNON
  SIM_cannon_analytic
    RUN_test
/Users/penn/trick.git/trick_sims
/Users/penn/trick.git/trick_sims/Ball
```

DP Tree

```
/Users/penn/CANNON
  SIM_cannon_analytic
    DP_Product
      DP_rt_frame.xml
      DP_rt_timeline.xml
      DP_rt_timeline_init.xml
      DP_rt_trickjobs.xml
      DP_rt_userjobs.xml
```

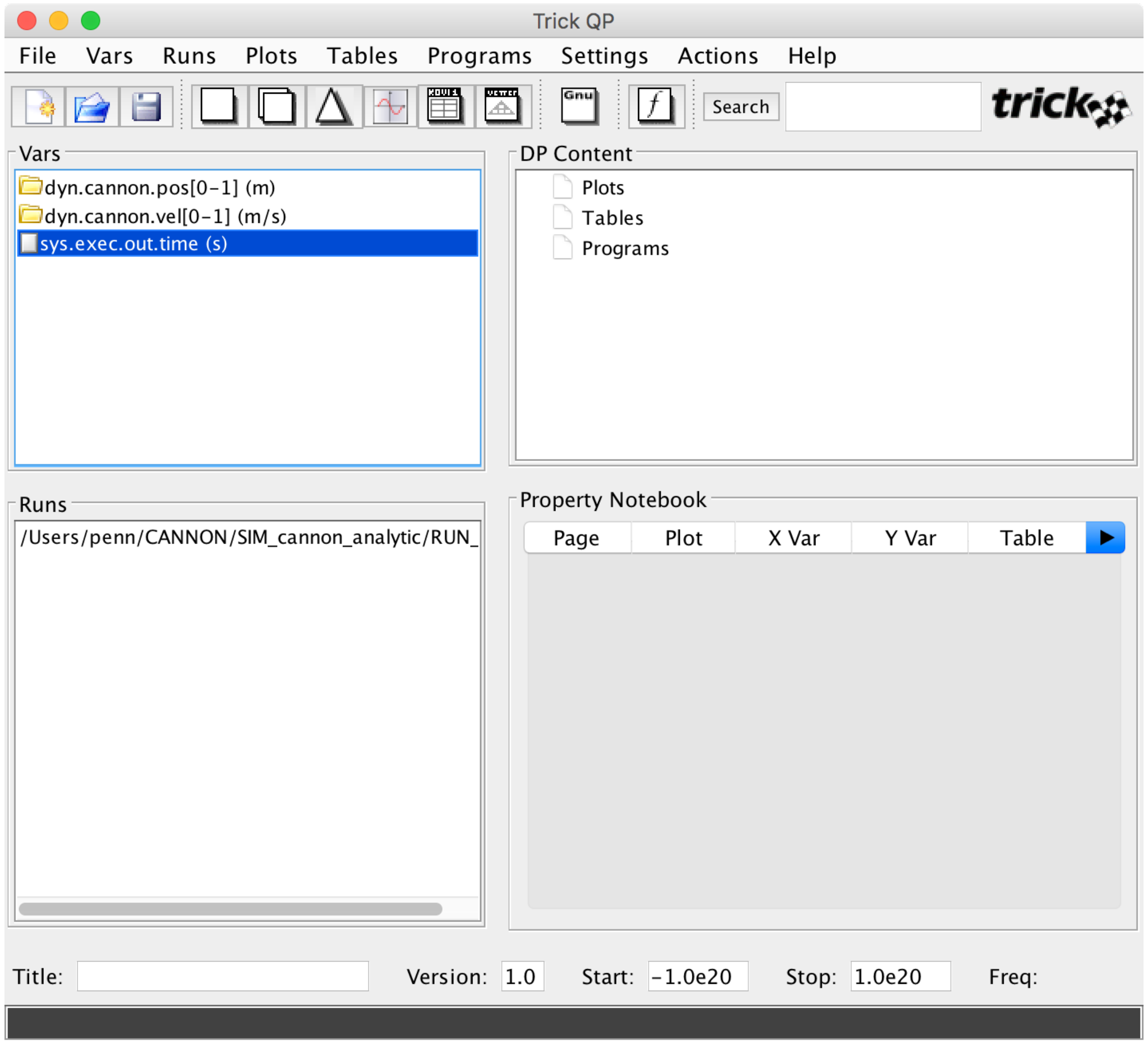
Run Selections

```
/Users/penn/CANNON/SIM_cannon_analytic/RUN_test
```

DP Selections

Version: 1.0 Start: -1.0e20 Stop: 1.0e20 Freq: 0.0

Plotting Recorded Data with *trick-qp*


Trick QP
File Vars Runs Plots Tables Programs Settings Actions Help
Search
Vars
DP Content
Runs
Property Notebook
Page Plot X Var Y Var Table

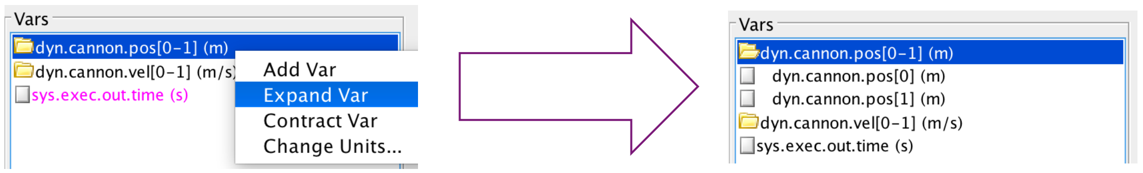
dyn.cannon.pos[0-1] (m)
dyn.cannon.vel[0-1] (m/s)
sys.exec.out.time (s)

Plots
Tables
Programs

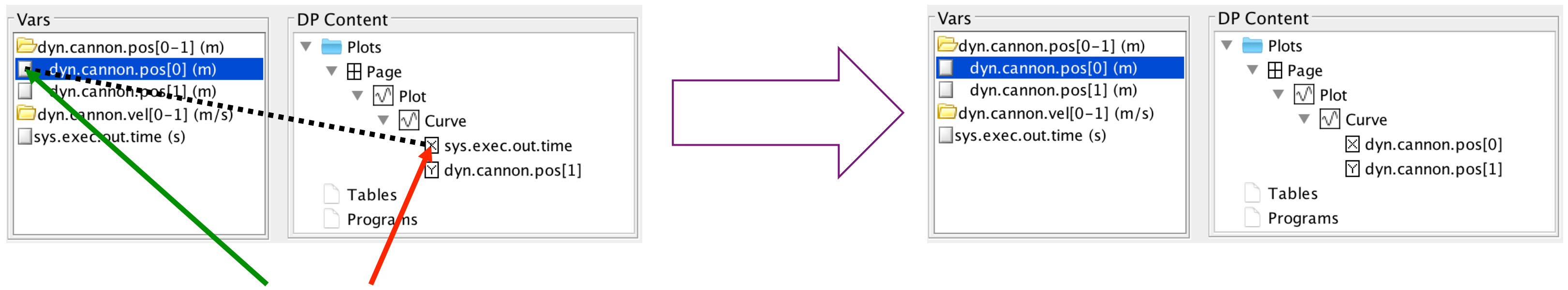
/Users/penn/CANNON/SIM_cannon_analytic/RUN_

Title: Version: 1.0 Start: -1.0e20 Stop: 1.0e20 Freq:

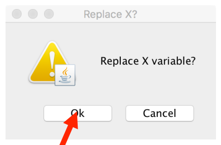
Telling *trick-qp* What to Plot



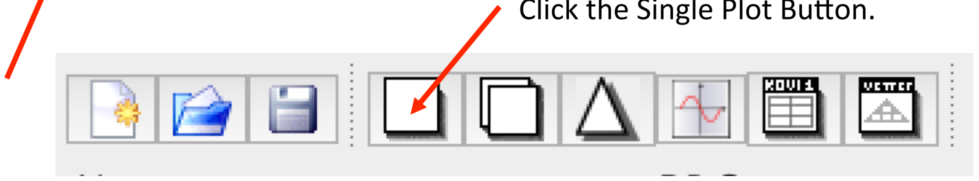
Click and drag **dyn.cannon.pos[0]** with left mouse button to the curves X coordinate.



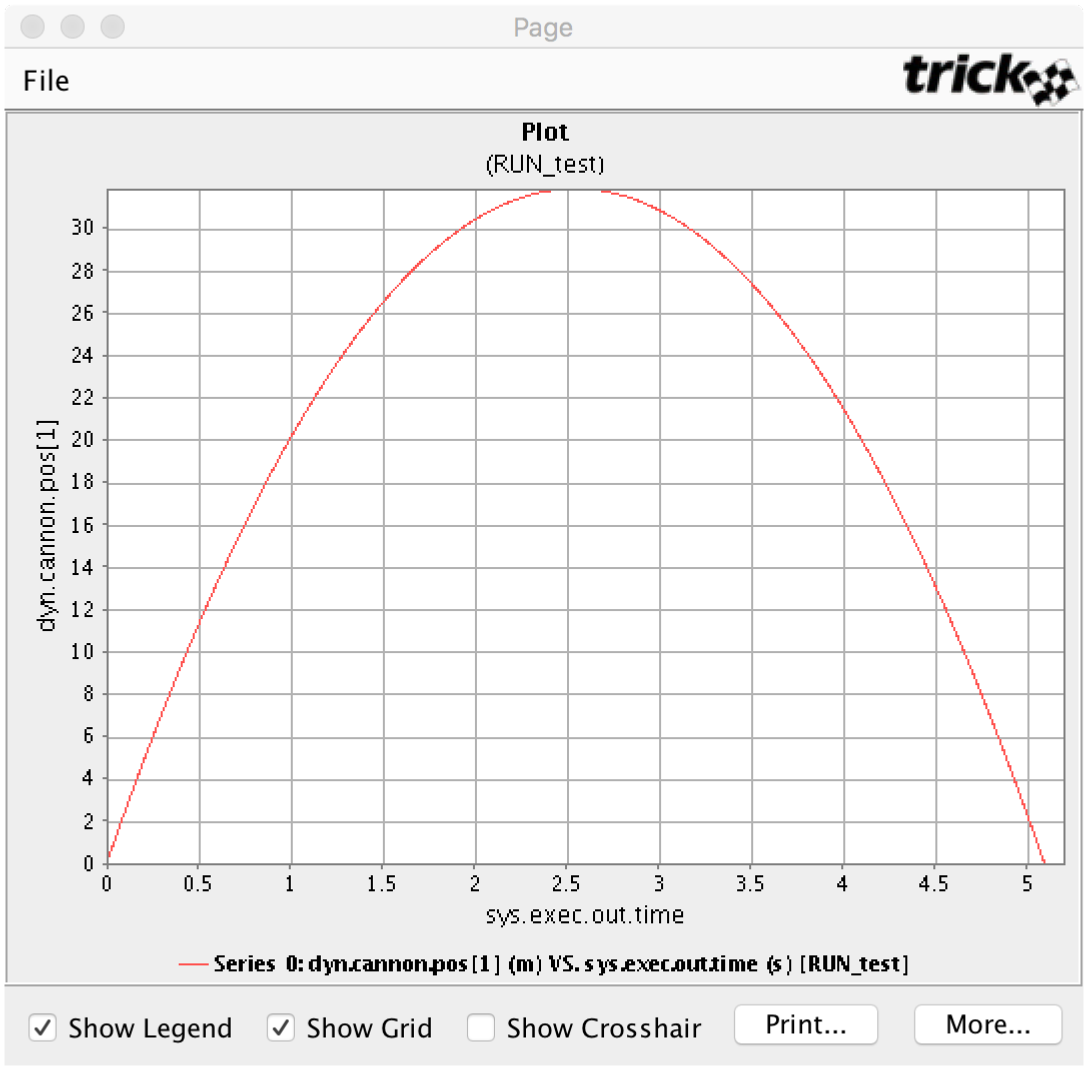
Click and drag from here to here.



Click the Single Plot Button.



Plotting Recorded Data

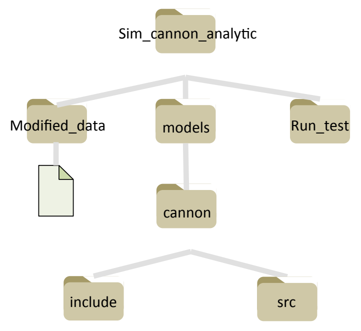


Running Real-time

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

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)
```



Specifying Real-time Operation

```
trick.real_time_enable()
```

Enables Trick real-time. In other words, it enables the periodic synchronization of simulation-time with real-time (“wall-clock-time”).

```
trick.exec_set_software_frame(0.1)
```

Specifies how often simulation-time is synchronized with real-time.

Once You're in Real-time

`trick.itimer_enable()`

Allows other processes to run while Trick is waiting for the beginning of the next software frame to start the simulation jobs. If interval timers are not used, Trick will spin waiting for the next beat.

`trick.exec_set_freeze_command()`

Puts the simulation in freeze mode at start-up

`trick.exec_set_enable_freeze()`

Allows a user to toggle the simulation mode between **`FREEZE`** and **`RUN`**.

`trick.sim_control_panel_set_enabled(True)`

Starts the sim control panel GUI.

`trick.frame_log_on()`

Log simulation performance data.

Running in Real-time

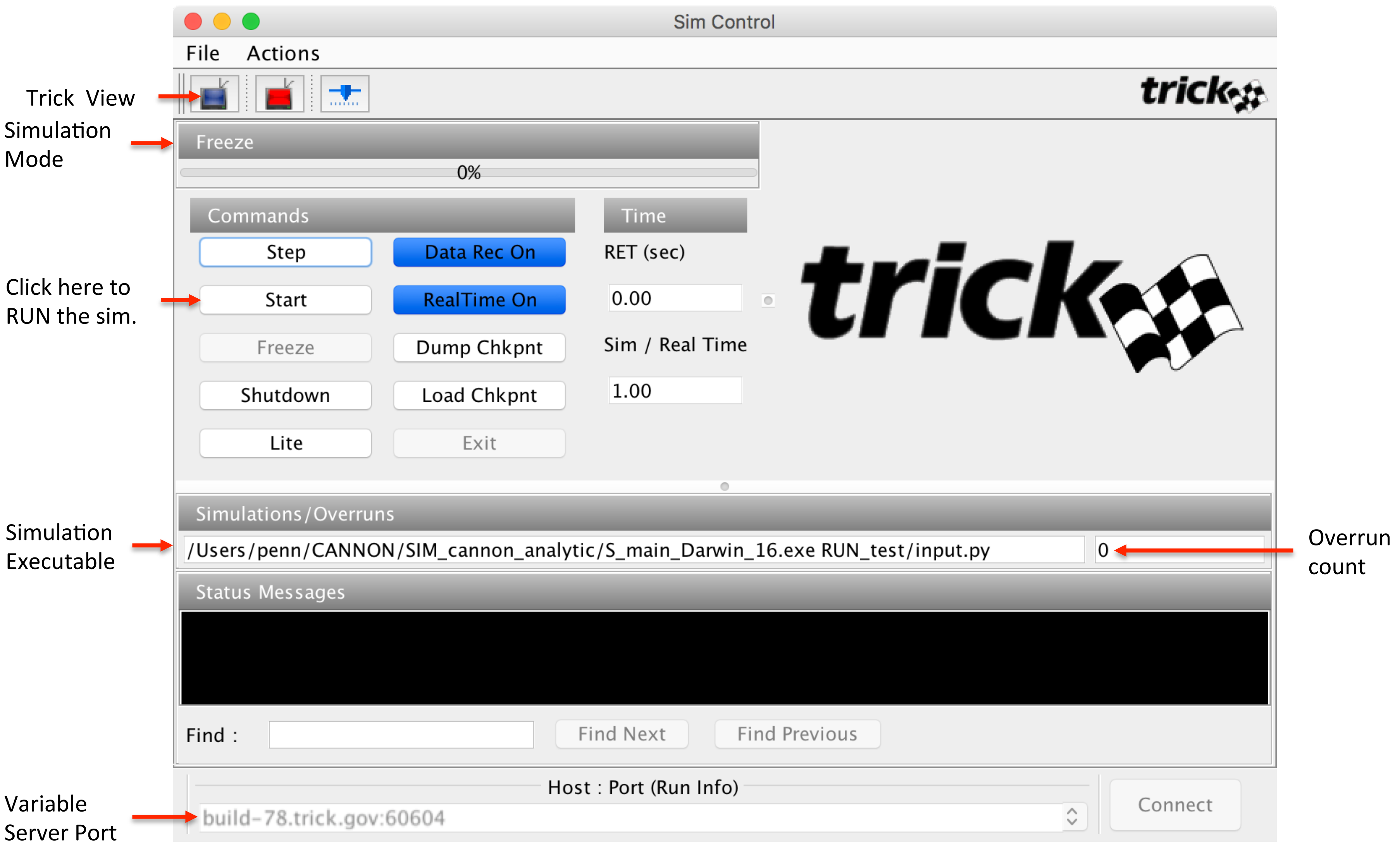
Update input.py

```
execfile("Modified_data/realtime.py")
execfile("Modified_data/cannon.dr")
trick.stop(5.2)
```

Re-run the simulation:

👉
```
% ./S_main*.exe RUN_test/input.py
```

Sim Control Panel

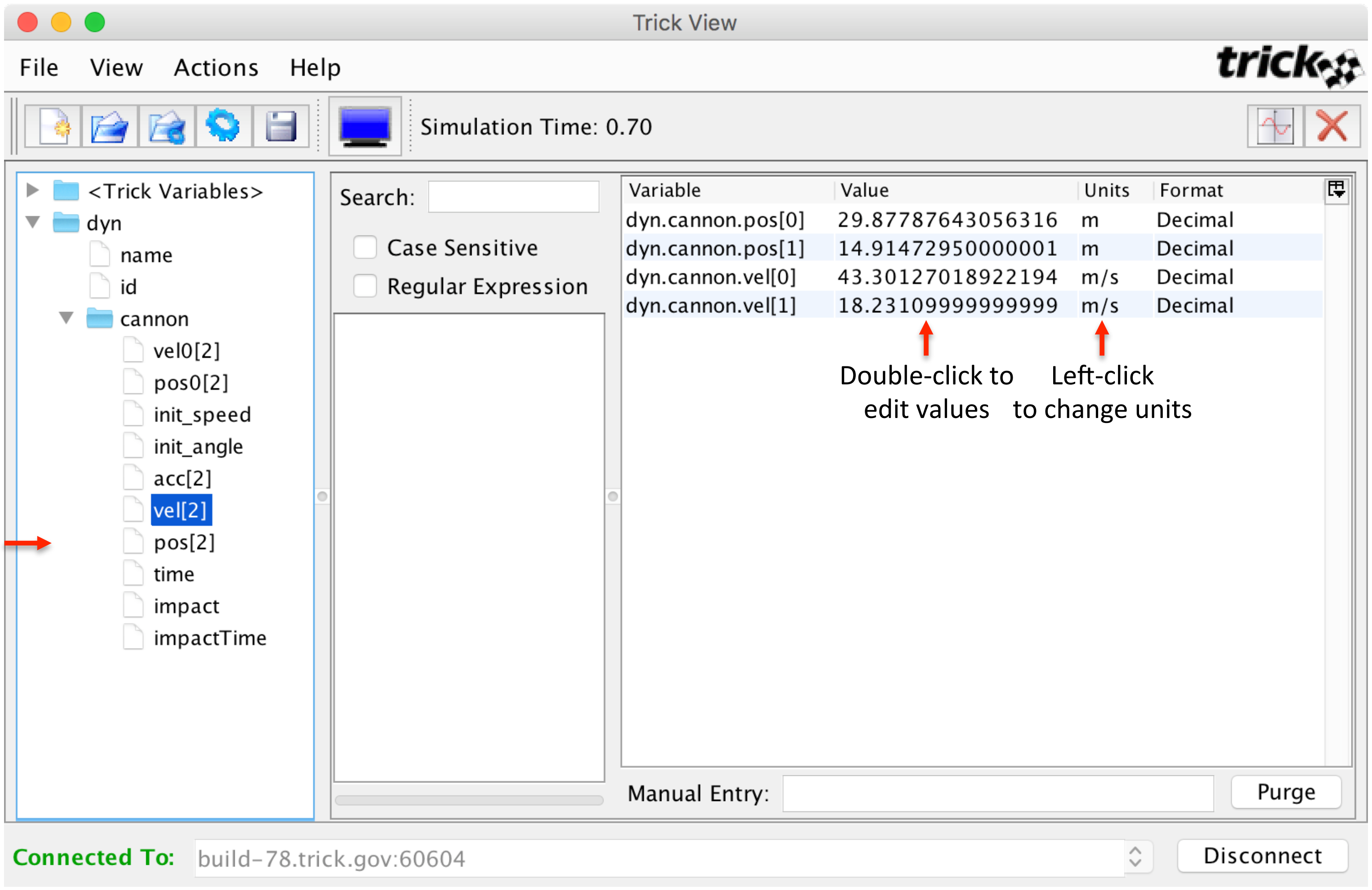


/Users/penn/CANNON/SIM_cannon_analytic/S_main_Darwin_16.exe RUN_test/input.py

0

build-78.trick.gov:60604

Trick View (TV)



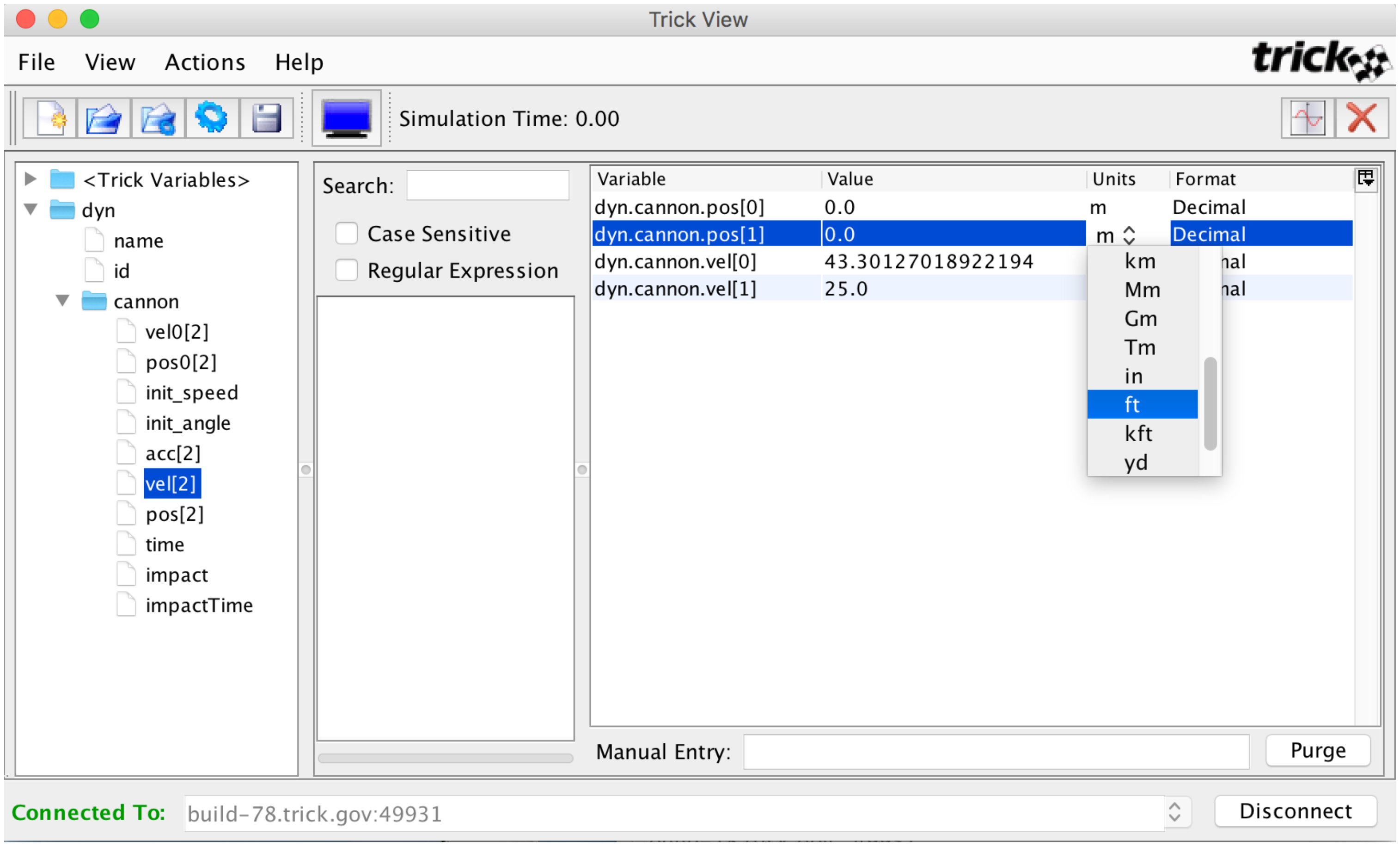
Variable	Value	Units	Format
dyn.cannon.pos[0]	29.87787643056316	m	Decimal
dyn.cannon.pos[1]	14.91472950000001	m	Decimal
dyn.cannon.vel[0]	43.30127018922194	m/s	Decimal
dyn.cannon.vel[1]	18.23109999999999	m/s	Decimal

Double-click to edit values

Left-click to change units

Double-click variables to select for viewing

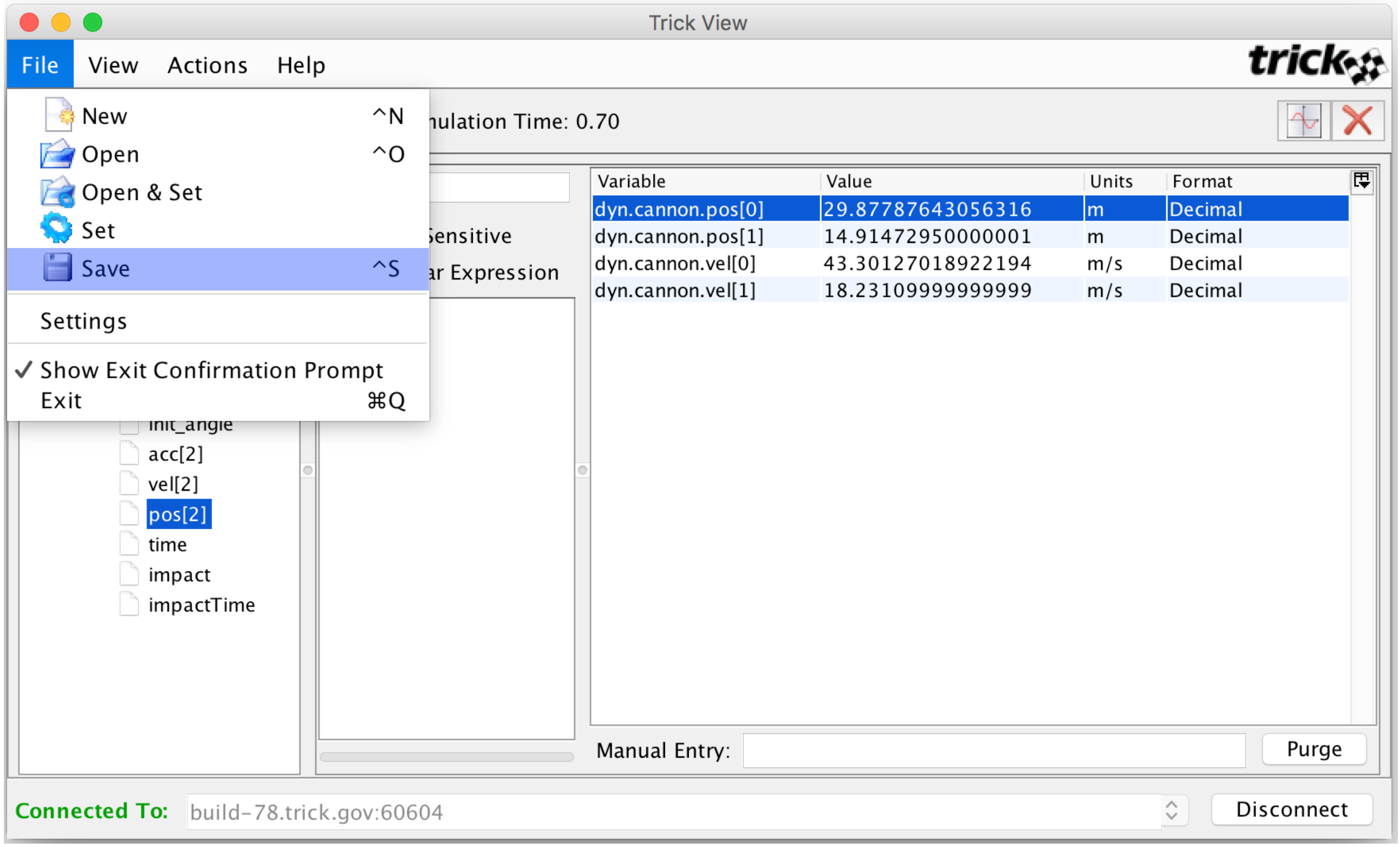
Trick View (TV)



Variable	Value	Units	Format
dyn.cannon.pos[0]	0.0	m	Decimal
dyn.cannon.pos[1]	0.0	m	Decimal
dyn.cannon.vel[0]	43.30127018922194		nal
dyn.cannon.vel[1]	25.0		nal

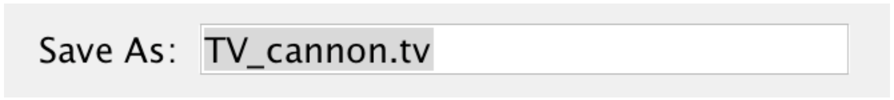
build-78.trick.gov:49931

Trick View (TV)



Variable	Value	Units	Format
dyn.cannon.pos[0]	29.87787643056316	m	Decimal
dyn.cannon.pos[1]	14.91472950000001	m	Decimal
dyn.cannon.vel[0]	43.30127018922194	m/s	Decimal
dyn.cannon.vel[1]	18.23109999999999	m/s	Decimal

Connected To: build-78.trick.gov:60604



Save As: TV_cannon.tv

Trick View (TV)

Update input.py

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

trick.trick_view_add_auto_load_file("TV_cannon.tv")
trick.stop(5.2)
```

Re-run the simulation:

👉
```
% ./S_main*.exe RUN_test/input.py
```

The sim will come up as before, but Trick View will also come up
With the previously saved variable set.

State Propagation with Numerical Methods

When Analytic Solutions Don't Exist

The simulation we've thus far created, relies on the fact that the cannon ball problem has an closed-form solution. From it, we can immediately calculate the cannon ball state (position, and velocity) at any arbitrary time. In real-world simulation problems, this will <u>rarely</u> be the case.

In this section, we're going to pretend that no analytic solution exists and model the cannon ball using **numeric-integration**.

Updating the Cannon Ball Sim to Use Numerical Integration

Rather than type everything again, we will first "tidy up" and copy the simulation. Then we'll modify it to use numeric integration.

To tidy up, execute the following:

```
% cd $HOME/trick_sims/SIM_cannon_analytic
% make spotless
```

Copy the sim directory, giving it a new name.

```
% cd ..
% cp -r SIM_cannon_analytic SIM_cannon_numeric
```

Job Classes for Numerical Integration

To provide simulation developers with a means of getting data into and out of these algorithms, Trick defines the following two job classes:

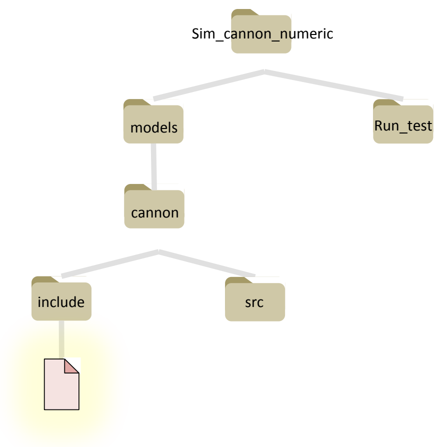
- **derivative** class jobs - for calculating the state time derivatives.
- **integration** class jobs - for integrating the state time derivatives from time t_{n-1} to t_n, to produce the next state.

A special **integ_loop** job scheduler coordinates the calls to these jobs **derivative** and **integration** jobs.

Interface Specifically for Numeric Job Functions

cannon_numeric.h

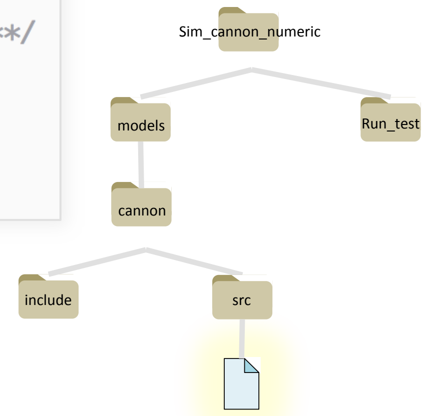
```
/*****************************************************************************
PURPOSE: ( Cannonball Numeric Model )
*****************************************************************************/

#ifndef CANNON_NUMERIC_H
#define CANNON_NUMERIC_H

#include "cannon.h"

#ifdef __cplusplus
extern "C" {
#endif
int cannon_integ(CANNON*) ;
int cannon_deriv(CANNON*) ;
#ifdef __cplusplus
}
#endif
#endif
```



Start of Cannonball Numeric Approach

cannon_numeric.c

```
/*****************************************************************************
  PURPOSE: ( Trick numeric )
*****************************************************************************/
#include <stddef.h>
#include <stdio.h>
#include "trick/integrator_c_intf.h"
#include "../include/cannon_numeric.h"
```



👉 Add the code snippet above to `cannon_numeric.c`

To this, we'll be soon our derivative and integration job functions.

Derivative Class Jobs

The purpose of a derivative class job is to generate a model’s time-derivatives, that is, it evaluates the right-hand side of the model’s differential equation.

For "F=ma" type models, derivative jobs calculate acceleration, by dividing force by mass.

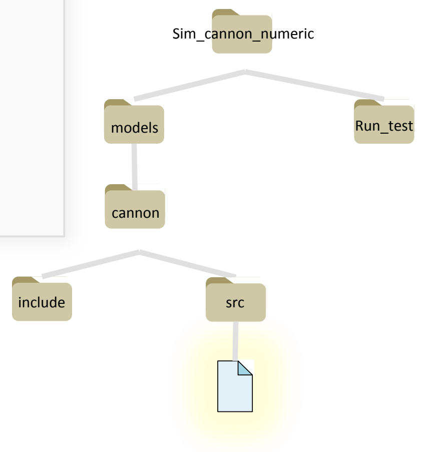
$$\vec{a}(t) = \vec{F}(t) / m(t)$$

All time dependent quantities from which acceleration is calculated should also be calculated in the derivative job.

In the corresponding integration class job, the acceleration is then integrated to produce velocity, and velocity is integrated to produce position.

Cannon Ball Derivative Job-function

```c
int cannon_deriv(CANNON* C) {

    if (!C->impact) {
        C->acc[0] = 0.0 ;
        C->acc[1] = -9.81 ;
    } else {
        C->acc[0] = 0.0 ;
        C->acc[1] = 0.0 ;
    }
    return(0);
}
```



👉 Add `cannon_deriv()` to `cannon_numeric.c`

Integration Class Jobs

The purpose of a integration class job is to integrate the derivatives that were calculated in the corresponding derivative jobs, producing the next simulation state from the previous state.

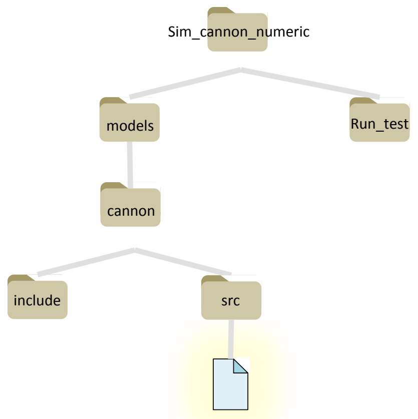
Integration jobs generally look very similar. That is because they are expected to do the same five things:

1. Load the state into the integrator.
2. Load the state derivatives into the integrator.
3. Call the integrate() function.
4. Unload the updated state from the integrator.
5. Return the value that was returned by the integrate() call.

Cannon Ball Integration Job-function

cannon_integ.c

```c
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);
}
```



👉 Add `cannon_integ()` to `cannon_numeric.c`

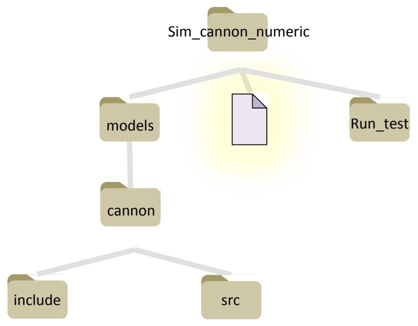
Updating the S_define File

👉 Update the `LIBRARY_DEPENDENCY` section:

```
REPLACE:  (cannon/src/cannon_analytic.c)
WITH:     (cannon/src/cannon_numeric.c)
```

👉 Update ##includes

```
REPLACE:  ##include "cannon/include/cannon_analytic.h"
WITH:     ##include "cannon/include/cannon_numeric.h"
```



👉 Update Scheduled Jobs

```
REPLACE:  (0.01, "scheduled") cannon_analytic( &cannon ) ;
WITH:
          ("derivative") cannon_deriv( &cannon) ;
          ("integration") trick_ret= cannon_integ( & cannon);
```

And one more thing …

Integration Loop Configuration

Producing simulation states by numerical integration requires that derivative and integration jobs be called at the appropriate rate and times. This requires a properly configured integration-scheduler. This is a two part process:

1. Create an integration-scheduler.
2. Select an integration algorithm for that scheduler.

Creating an Integration Scheduler

An integration scheduler is instantiated in the S_define. It takes the form:

```
IntegLoop integLoopName ( integrationTimeStep ) listOfSimObjectNames ;
```

Jobs within named simObjects that are tagged “`derivative`" or "`integration`” will be dispatched by the associated integration scheduler.

Selecting an Integrator

In the input file, call the IntegLoop **getIntegrator()** method to specify the integration algorithm of choice and the number of state variables to be integrated.

```
integLoopName.getIntegrator( algorithm, N );
```

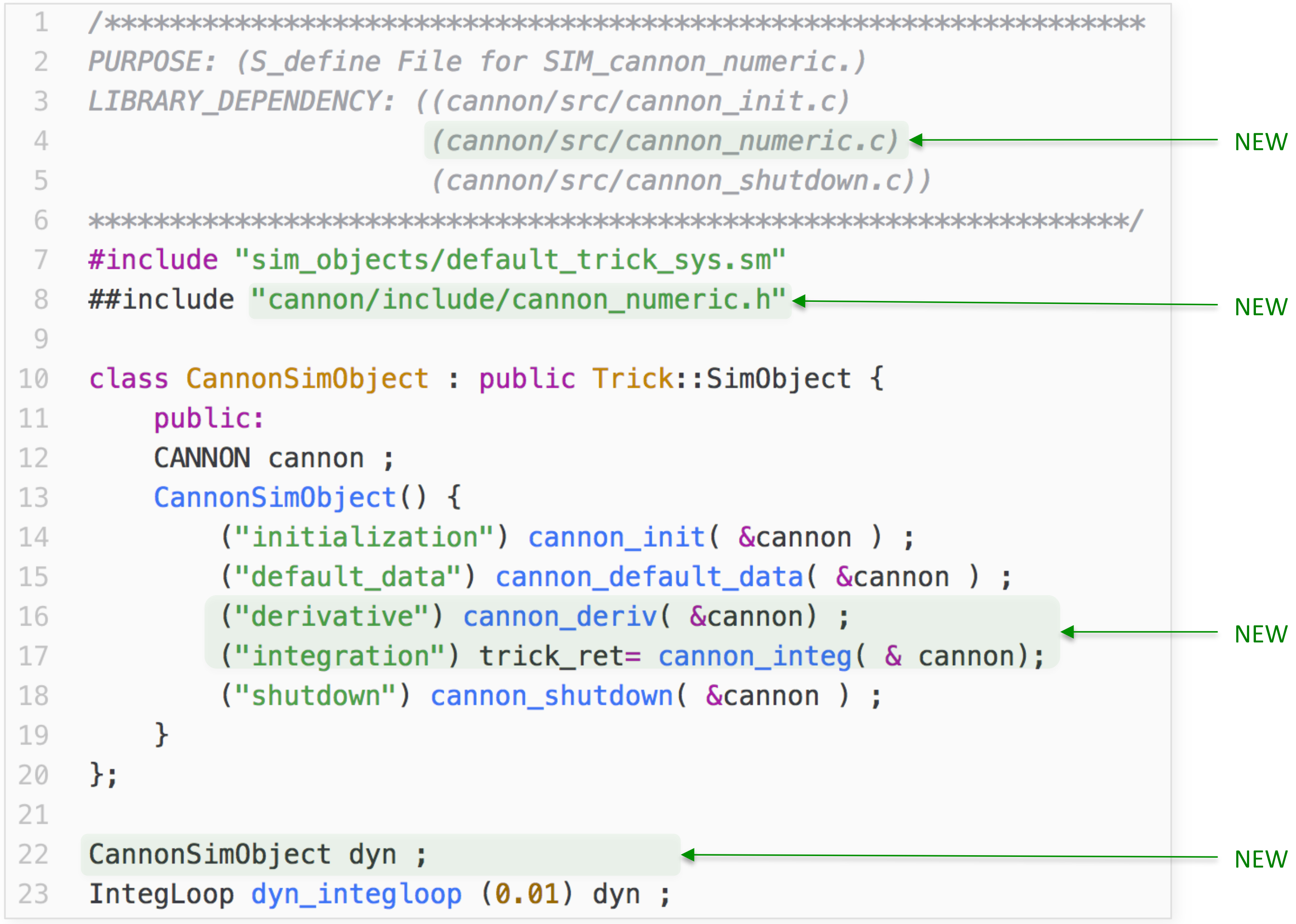
algorithm is an enumeration value that indicates which numerical integration algorithm to use, such as: `trick.Euler`, `trick.Runge_Kutta_2`, `trick.Runge_Kutta_4, etc*`.

N is the number of state variables to be integrated.

* A complete list of available algorithms can be seen Integrator.hh, in ${TRICK_HOME}/include/trick/Integrator.hh .

S_define for Numeric Sim

S_define



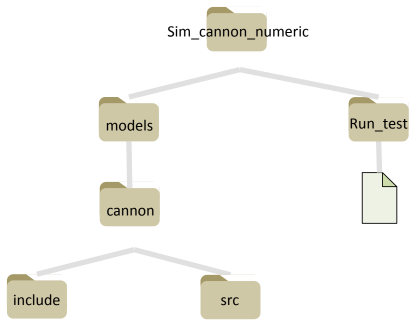
```
/*****************************************************************
PURPOSE: (S_define File for SIM_cannon_numeric.)
LIBRARY_DEPENDENCY: ((cannon/src/cannon_init.c)
                     (cannon/src/cannon_numeric.c)
                     (cannon/src/cannon_shutdown.c))
*****************************************************************/
#include "sim_objects/default_trick_sys.sm"
##include "cannon/include/cannon_numeric.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);
        ("shutdown") cannon_shutdown( &cannon ) ;
    }
};

CannonSimObject dyn ;
IntegLoop dyn_integloop (0.01) dyn ;
```

Input File for Numeric Sim

input.py

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

dyn_integloop.getIntegrator(trick.Runge_Kutta_4, 4)

trick.stop(5.2)
```

NEW



👉 Update `input.py` as shown.

Running the Numeric Sim

```
% cd $HOME/trick_sims/SIM_cannon_numeric
% trick-CP
```

```
% ./S_main*.exe RUN_test/input.py
```

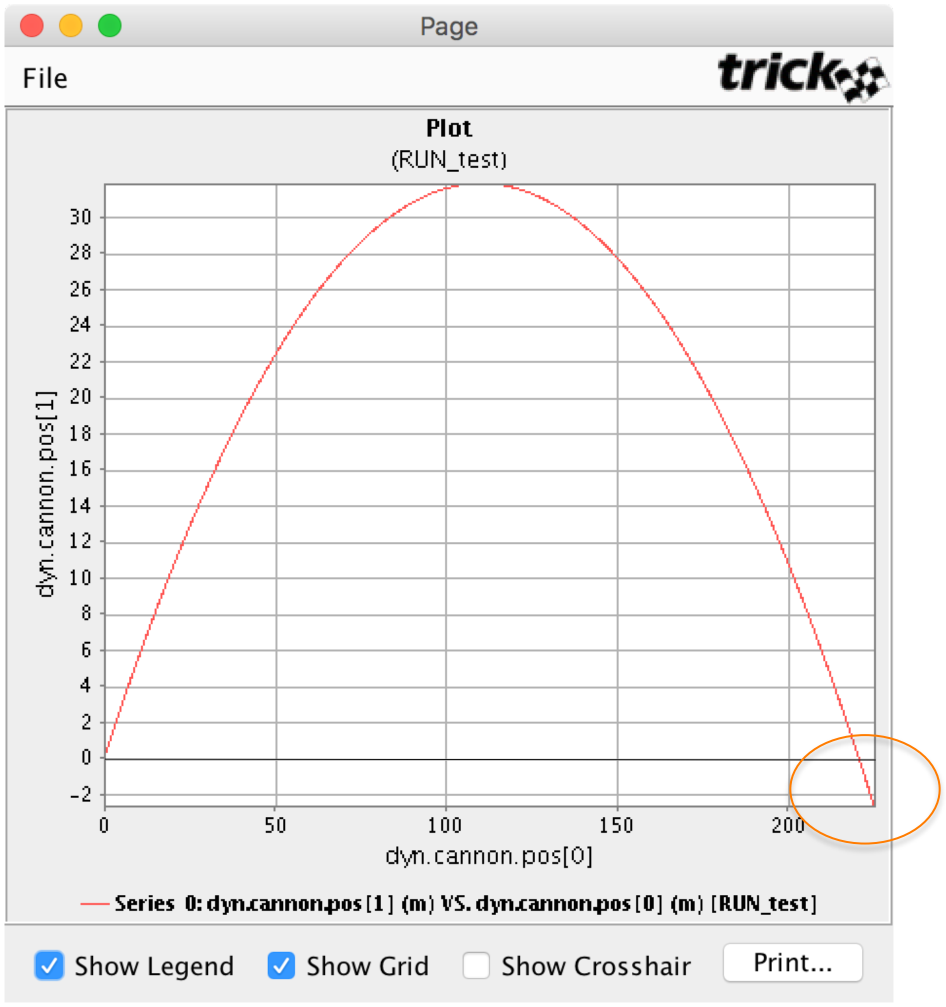
We get:

```
========================================
      Cannon Ball State at Shutdown
t = 5.2
pos = [225.166604984, -2.631200000]
vel = [43.301270189, -26.012000000]
========================================
```

DON'T PANIC!

It's not the same. That's because didn't stop our sim at impact-time. We will.

Running the Numeric Sim



In our analytic sim, we stopped updating our cannonball state at impact, when we crossed *y=0*, at *t=5.096839*.

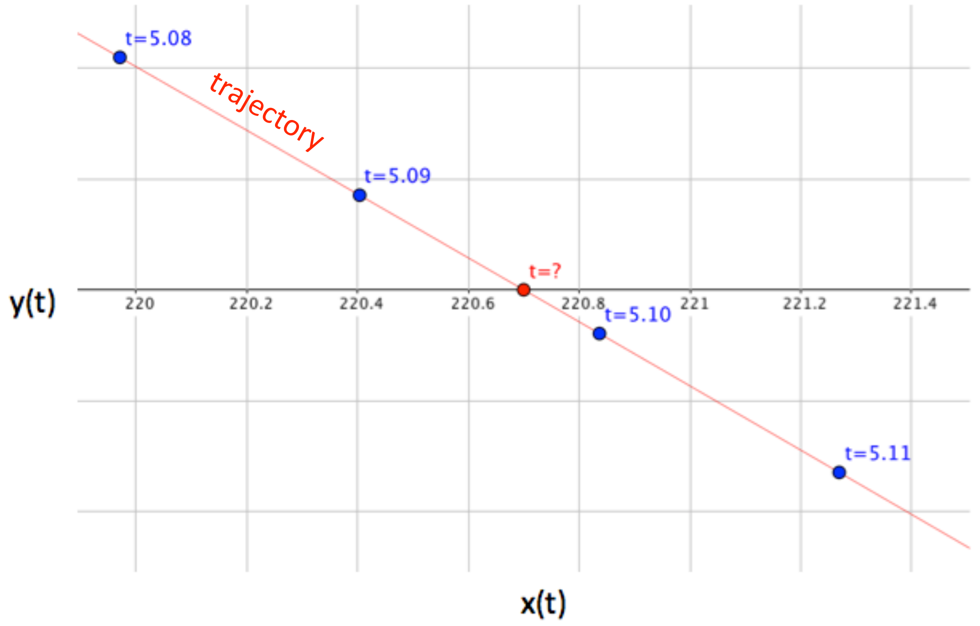
In our numeric sim, didn't detect impact, and continued to update the cannonball state until *t = 5.2*.

So, what about the impact-time?

Remember, we're pretending that we don't have an analytical solution. So, we can't use our time-of-impact equation.

Dynamic Events

We need a numerical method to determine the precise time of impact, *t* when *y(t)=0*.



In Trick, we call this type of occurrence, when our simulation state is at some boundary that we've defined, a **dynamic-event**. To find dynamic-events, we use **dynamic-event** jobs.

Dynamic Event Jobs

The job scheduler calls dynamic-event jobs, after each integration step:

- To detect when the simulation state crosses a user-defined event boundary,
- To take control of integration, to find the exact event-state and time, and
- To perform some action as a result of finding the event-state.

It does this using the Trick's regula_falsi() function and REGULA_FALSI data-type to implement the **False Position Method**.

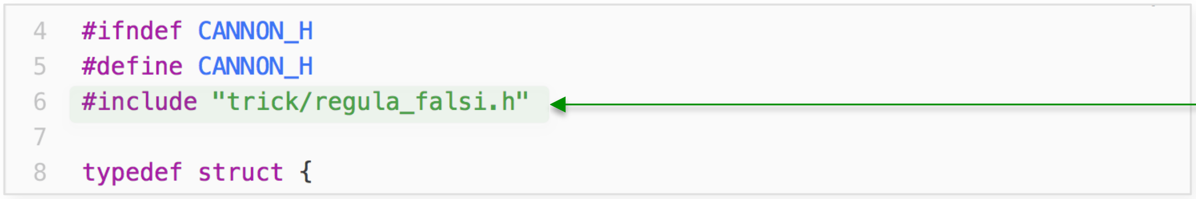
Finding Events with *regula_falsi()*

The regula_falsi() function is the heart of a dynamic event function. It's job is to:

- Monitor the simulation state produced by each integration step,
- Detect when the state crosses a specified event boundary, and
- Guide Trick's integration scheduler to find that event.

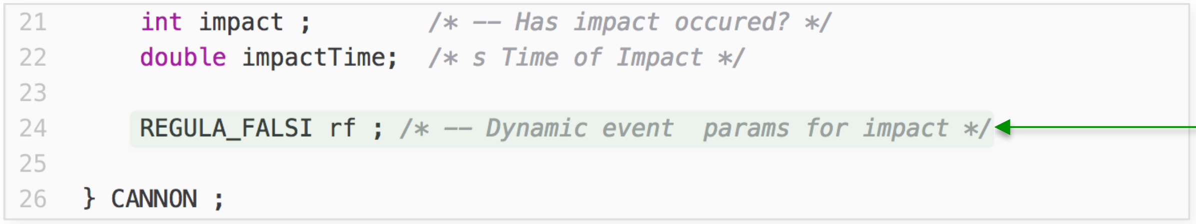
Progress toward finding the event state is recorded in a `REGULA_FALSI` variable.

Updates to cannon.h



```
#ifndef CANNON_H
#define CANNON_H
#include "trick/regula_falsi.h"

typedef struct {
```

NEW

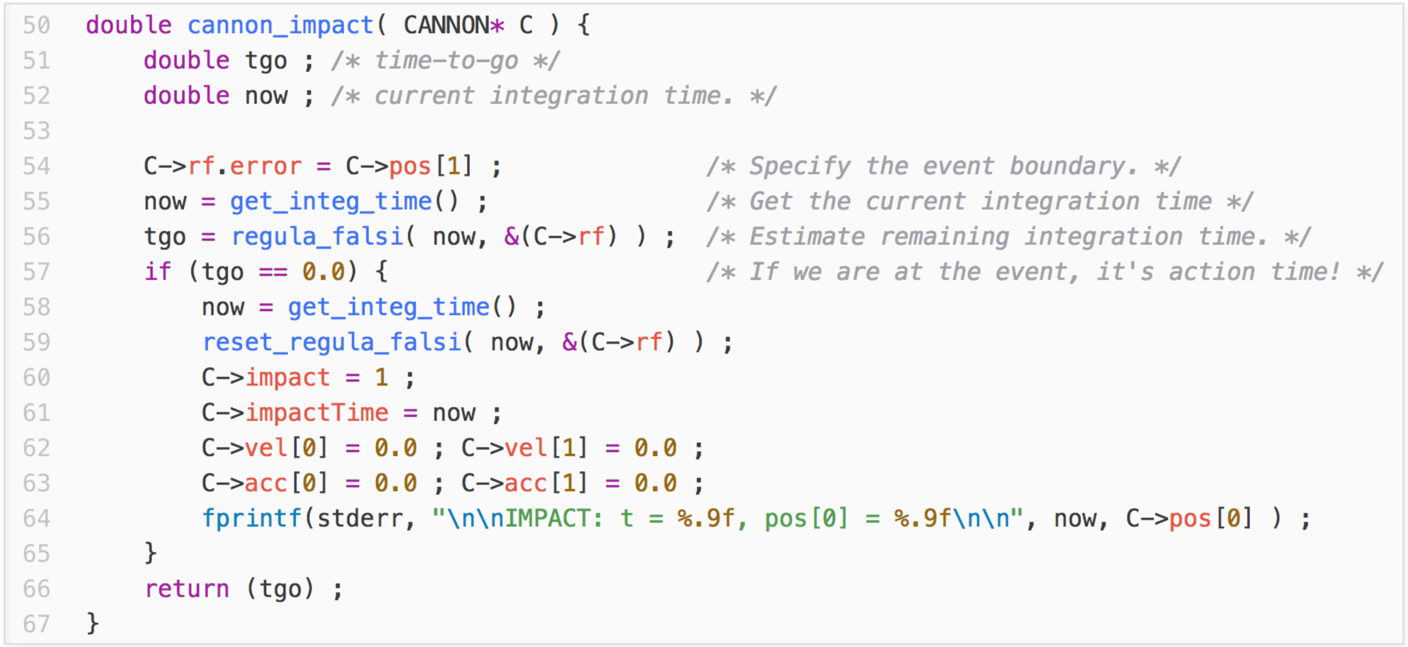


```
    int impact ;          /* -- Has impact occured? */
    double impactTime;  /* s Time of Impact */

    REGULA_FALSI rf ; /* -- Dynamic event  params for impact */

} CANNON ;
```

NEW

👉 Update cannon.h as shown.

Dynamic Event Job Function: *cannon_impact()*

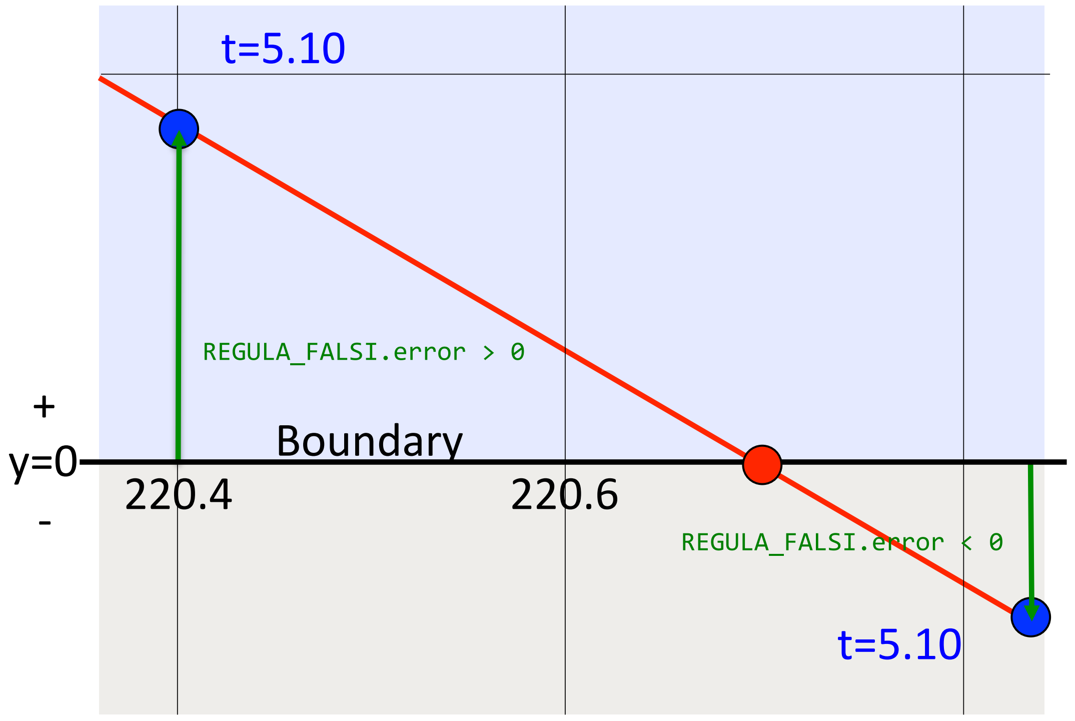


```c
double cannon_impact( CANNON* C ) {
    double tgo ; /* time-to-go */
    double now ; /* current integration time. */

    C->rf.error = C->pos[1] ;              /* Specify the event boundary. */
    now = get_integ_time() ;               /* Get the current integration time */
    tgo = regula_falsi( now, &(C->rf) ) ; /* Estimate remaining integration time. */
    if (tgo == 0.0) {                      /* If we are at the event, it's action time! */
        now = get_integ_time() ;
        reset_regula_falsi( now, &(C->rf) ) ;
        C->impact = 1 ;
        C->impactTime = now ;
        C->vel[0] = 0.0 ; C->vel[1] = 0.0 ;
        C->acc[0] = 0.0 ; C->acc[1] = 0.0 ;
        fprintf(stderr, "\n\nIMPACT: t = %.9f, pos[0] = %.9f\n\n", now, C->pos[0] ) ;
    }
    return (tgo) ;
}
```

👉 Add this function, to the bottom of `cannon_numeric.c`

Specifying an Event Boundary – `REGULA_FALSI.error`

`REGULA_FALSI.error` – how far and on which side of the boundary is the cannon ball. We set rf.error to the y-coordinate of the ball.



Specifying an Event Boundary – `REGULA_FALSI.mode`

`REGULA_FALSI.mode` – enumeration value that constrains an event to a particular direction of boundary crossing.

- **`Increasing`** - specifies that an event occurs only when the boundary is crossed from negative to positive.
- **`Decreasing`** - specifies that an event occurs only when the boundary is crossed from positive to negative.
- **`Any`** - (default) specifies that an event occurs when the boundary is crossed from either direction.

Calling *regula_falsi()*

The regula_falsi function estimates the amount of time until `REGULA_FALSI.error` reaches `0`, that is, when the boundary will be crossed.

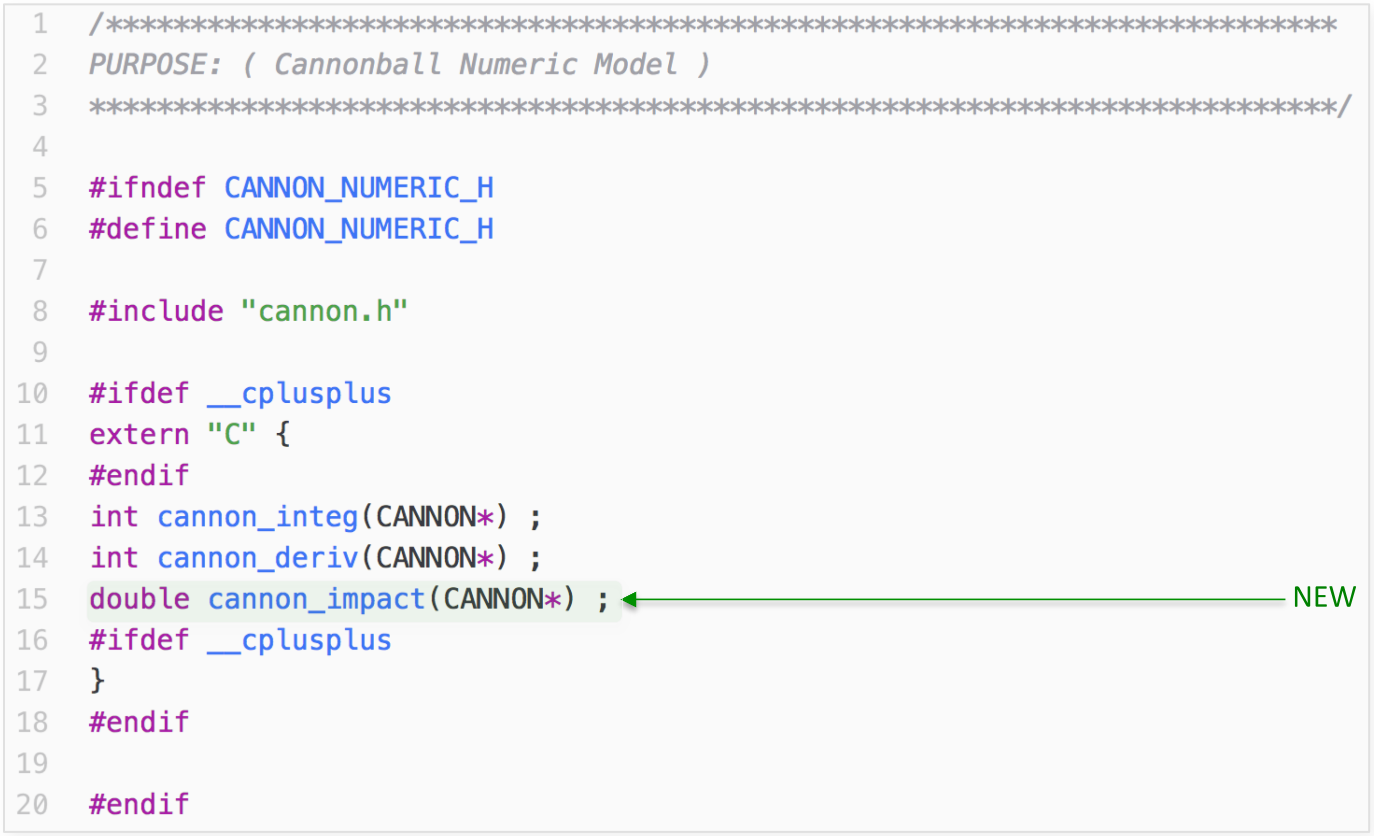
```
double regula_falsi( currentIntegrationTime, regulaFalsi_p );
```

currentIntegrationTime – *from* `get_integ_time()`

regulaFalsi_p – *pointer to* `REGULA_FALSI` *object.*

Returns – an estimate of the amount of time until the event.

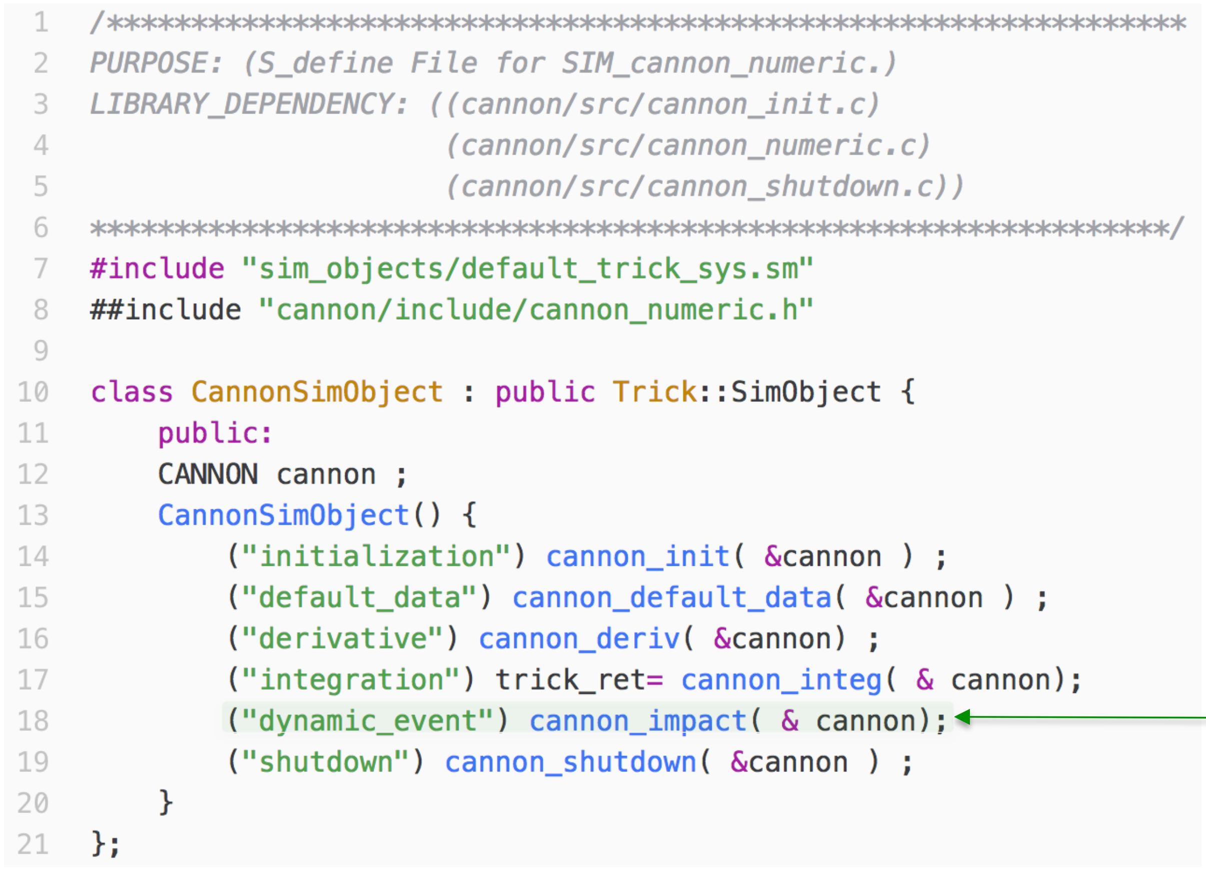
Update `cannon_numeric.h`



```
/*************************************************************************
PURPOSE: ( Cannonball Numeric Model )
**************************************************************************/

#ifndef CANNON_NUMERIC_H
#define CANNON_NUMERIC_H

#include "cannon.h"

#ifdef __cplusplus
extern "C" {
#endif
int cannon_integ(CANNON*) ;
int cannon_deriv(CANNON*) ;
double cannon_impact(CANNON*) ;
#ifdef __cplusplus
}
#endif

#endif
```

NEW

Update S_define for Numeric Sim



```
/*****************************************************************
PURPOSE: (S_define File for SIM_cannon_numeric.)
LIBRARY_DEPENDENCY: ((cannon/src/cannon_init.c)
                     (cannon/src/cannon_numeric.c)
                     (cannon/src/cannon_shutdown.c))
*****************************************************************/
#include "sim_objects/default_trick_sys.sm"
##include "cannon/include/cannon_numeric.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);
        ("dynamic_event") cannon_impact( & cannon);
        ("shutdown") cannon_shutdown( &cannon ) ;
    }
};
```

NEW

Running the Completed Numeric Sim

👉 Rebuild SIM_Cannon_numeric.

👉 Run it.

We get:

```
IMPACT: t = 5.096839959, pos[0] = 220.699644186

========================================
      Cannon Ball State at Shutdown
t = 5.2
pos = [220.699644186, 0.000000000]
vel = [0.000000000, 0.000000000]
========================================
```

The **same** answer we got with our analytic sim!

Congratulations!

You’ve completed the Basic Trick Tutorial Review!