Solution Manual

to accompany

JEOD Training Course

for JEOD version 5.0

Gary Turner (updated by Christopher Sullivan)
Odyssey Space Research
December, 2016 (updated July, 2022)

Introduction

This manual is to be used to assist users with developing solutions to the exercises contained within the course. For simulation-based exercises, solution files have been provided to illustrate one possible way to construct the S_define and data files necessary to meet the exercise objective. This manual is intended to be a companion-guide to those solutions, providing a step-by-step explanation of the structure of those solution files.

The provided solutions are intended to be used for comparison to the student's own solutions, and this manual is intended to provide an explanation for why the solution was constructed the way it was. The provided solutions are not unique, so where differences exist between your solutions and the provided solutions, this manual can be used to discern whether the differences are simply stylistic, or more substantive.

The manual also provides some important lessons learned that result from the successful completion of the exercises, as well as suggestions for data comparisons to verify that the developed simulations are functioning as intended.

Line numbers refer to line numbers in the files contained within the provided solutions. Note that line numbers should not be entered as text, and depending on the editor you use, may not be visible in the files themselves. Note also that this manual will go through the solutions files by grouping like terms together; that may not be the same as the structure of the solution files, so elements of a solution file may appear out of order in this manual.

Some words of caution:

- This manual does not step through the solutions line-by-line, so it should not be used to construct a solution line-by-line. Remember, this manual is to help *explain* why the solution files represent a solution to the problem at hand, not to construct them for you.
- The explanatory notes constitute explanations of how a specific solution meets the requirement. They do not provide the methodology by which the solution was generated, so using this manual as an instructional guide for how to use JEOD is probably not a good idea.
- Solutions are written to support Trick-19. Note that the units specification has changed between Trick versions and Trick-19 can interpret units as specified for previous versions of Trick. Consequently, there are instances where warning messages regarding units are seen when running in Trick-19. These can be ignored.
    - e.g. `Units converted from [d] to [degree] Modified_data/vehicle_state.py:12`

Bold-faced text in this document highlights particularly important points within the solutions, it does not reference some magical font in the solutions themselves.

Table of Contents

Exercise 1 Familiarity with the S_define....5
Exercise 2 Investigating Abstract Classes....6
Exercise 3 Tracing Inheritance....7
Exercise 4 Identifying the Integration Algorithm....8
Exercise 5 Simple simulation....9
Exercise 6 Simple Simulation, adding clocks....12
Exercise 7 Two-vehicle Simulation....16
Exercise 8 Creating a Simple Spherical-Earth Orbiter....18
Exercise 9 Creating a Simulation with Two Earths....19
Exercise 10 Complications of Adding a General Gravity Model....19
Exercise 11 Creating a Spherical-Earth Orbiter using a Non-spherical Gravity model....19
Exercise 12 Creating a Non-spherical-Earth Orbiter....21
Exercise 13 Multiple Gravitational Fields....23
Exercise 14 Lunar Orbiting Vehicles....28
Exercise 15 Gravity Gradient....29
Exercise 16 Solid Body Tides....31
Exercise 17 Interpretation of Variables....33
Exercise 18 Adding Mass Points and Internal Reference Frames....34
Exercise 19 Interpreting the Mass Tree - Depleting a Tank Mass....37
Exercise 20 Exercise with Mass Points and Body Ref Frames....40
20.1 Sensors, Body Reference Frames, and Mass Points....41
Exercise 21 A New Two-vehicle Simulation....43
21.1 Mass Properties....43
21.2 Setting the Attach and Detach Processes....44
21.3 Setting the Vehicle States and Testing....44
Exercise 22 Review of Exercise 19....48
Exercise 23 Initializing with Orbital Elements....49
Exercise 24 Initializing from LVLH....50
Exercise 25 Add an Atmosphere....53
Exercise 26 Setting up a Surface Model....54
Exercise 27 Add Aerodynamic Drag....55
Exercise 28 Add Radiation Pressure....58
Exercise 29 Utilizing the Contact Model....62
Exercise 30 Articulating Surfaces....66
30.1 Re-orienting an Array....66
30.2 Verifying the Re-orientation....67
Exercise 31 Smooth Rotation, with Radiation Pressure....68
Exercise 32 Simple Simulation of Vehicle with Thrusters....71
Exercise 33 Revisiting an Orbital-Elements Initialized Simulation; Output Orbital-Elements....72
Exercise 34 Derived States Exercise - Orbital Elements and Planet Fixed States....74
Exercise 35 Derived States Exercise – Relative State....76
Exercise 36 Derived States Exercise - Define a LVLH Reference Frame....78
Exercise 37 Derived States Exercise - Relative LVLH State....78
Exercise 38 Solar Beta....80

Exercise 39 Managing Relative States with the Relkin Manager.....82
Exercise 40 Managing Relative States with the Relkin Manager, part II.....84
Exercise 41 Multiple Integration Groups.....85
Exercise 42 Cis-lunar 2-vehicle Simulation and Ephemerides Rate Investigation.....87

Exercise 1 Familiarity with the S_define

1. There are 3 simulation objects (besides the system modules, remember, we are ignoring trick_sys and jeod_sys):
   a. jeod_time
   b. dynamics
   c. vehicle

   Note that the *default_priority_settings.sm* file does not define a simulation-object.
2. No solution
3.
   a. First run are the default data class jobs:
      i. time_converter_tai_ut1_default_data.initialize(&time_converter_tai_ut1)
   b. Next is the input file reading
   c. Then all derivative class and initialization class jobs:
      i. time_manager.register_type (time_tai)
      ii. time_manager.register_converter (time_converter_dyn_tai)
      iii. time_manager.register_type (time_ut1)
      iv. time_manager.register_converter (time_converter_tai_ut1)
      v. time_manager.initialize (&time_manager_init)
      vi. time_ut1.calendar_update(exec_get_sim_time())
      vii. dyn_manager.initialize_model (dyn_manager_init, time_manager);
      viii. dyn_body.initialize_model(dyn_manager);
         - from the vehicle object; it has P_ENV so comes before the next call
      ix. dyn_manager.initialize_simulation ( );
         - from the dynamics object; even though it is encountered before the previous call, it is set to a later priority so is executed later.
   d. Then the scheduled jobs at t = 0.0
      i. time_manager.update (exec_get_sim_time())
         - This runs at the DYNAMICS rate, twice every second
      ii. time_ut1.calendar_update (exec_get_sim_time())
         - This runs at the CALENDAR_INTERVAL rate, once every second
   e. Then the scheduled jobs at t = 0.5
      i. time_manager.update (exec_get_sim_time())
   f. Then the scheduled jobs at t = 1.0
      i. time_manager.update (exec_get_sim_time())
      ii. time_ut1.calendar_update (exec_get_sim_time())

Exercise 2 Investigating Abstract Classes

1. The following classes are abstract:
   - *JeodBaseTime*, with pure virtual method *initialize_initializer_time*.
   - *TimeConverter*, with pure virtual method *initialize.*
   - *TimeStandard*, with pure virtual method *set_epoch.*
2.
   - *TimeDyn*, *TimeStandard* and *TimeUDE* inherit from *JeodBaseTime.*
   - *TimeConverter** all inherit from *TimeConverter.*
   - *TimeGMST*, *TimeGPS*, *TimeTAI*, *TimeTDB*, *TimeTT*, *TimeUT1*, and *TimeUTC* all inherit from *TimeStandard*.
3. *TimeStandard* cannot be instantiated. Even though it defines an instance of its parent's pure virtual function (*initialize_initializer_time*), it adds one of its own (*set_epoch*), and so is an abstract class. All of its subclasses define the *set_epoch* function, and all of these subclasses can be instantiated.

Exercise 3 Tracing Inheritance

- *subject* is located in *BodyAction*, and is a *MassBody*.
- *core_properties* is located in *MassBody*, and is a *MassProperties*.
- *inertia* is located in *MassProperties*, and is described as being:
  - *Inertia tensor of the subject mass element about the subject mass element's center of mass, expressed in subject mass element body coordinates.*

Exercise 4 Identifying the Integration Algorithm

This exercise is not only a test of ability to identify integrators, but a lesson in why it is important to follow good practice and to cleanup unused code. The answer is the Adams-Bashforth-Moulton 4th order. If you got this one correct, very well done, there are some commonly-found misleading practices included in this exercise.

Notice that in the S_define, 3 different integrator constructor header files are included (note – those header file includes are not necessary, but it is good practice to add them), but only 1 instance is created – that of the Euler Integrator. But, the answer is NOT the Euler Integrator.

Remember that the instance itself is insufficient, it must be assigned to the *DynManagerInit* object's *integ_constructor* variable in the input file.

Taking a look at the input file, we see that an instance of an RK4 (Runge-Kutta 4th order) integrator is created, and the *integ_constructor* value is correctly assigned. However, the answer is NOT the RK4 integrator.

Further down in the input file, we have

```
50 execfile("Modified_data/data.py")
```

This is another bad practice – data files should be named something informative. Had this been named integrator.py, it may have been more obvious that it contained information about the integrator.

In this file, the *integ_constructor* value is reassigned to provide the ABM (Adams-Bashforth-Moulton 4th order, according to the API document) integrator. The *integ_constructor* is not re-defined after this point, so this is the answer.

To be completely correct, you would have to have read the integration documentation to have learned that the ABM integrator initially uses the RK4 integrator to generate the first few data points, so it could be argued that the absolutely correct answer is RK4 initially, followed by ABM.

Exercise 5 Simple simulation

1. First, the basics: must define the calling rate; include the trick and JEOD system objects; because we are using simulation modules, the *default_priority_settings* file; and an integration statement:

```
 8 #define DYNAMICS      1.0
11// Include the default trick object
12 #include "sim_objects/default_trick_sys.sm"
13
14 // Include the default jeod object
15 #include "jeod_sys.sm"
16 #include "default_priority_settings.sm"
...
28 IntegLoop sim_integ_loop (DYNAMICS) dynamics;
```

   Next, the three simulation objects of interest:

   barebones-time:

```
19 #include "time_dyntime_only.sm"
```

   dynamics that includes integration capability *(dynamics_init_only.sm* does not meet the integration requirement):

```
22 #include "dynamics.sm"
```

   vehicle

```
25 #include "vehicle_basic.sm"
```

2. Edit the following lines:

   a.
```
25 vehicle.mass_init.properties.mass = 500.0
```

   b. Note – there are two position values in the input file, the state is set in the *trans_init* and *rot_init* values. The orientation and angular rate values presented below are not definitive; there are multiple ways of completing parts iii and iv. This solution starts with a 180 degree rotation on the z-axis, decreasing at 36 degrees per second. The orientation is configured relative to the Space.inertial frame, the angular rate is in the composite_body frame. Because the body frame is only a z-rotation from the inertial frame, the two frames have identical z-axes, so the angular-rate z-axis is parallel to the orientation z-axis.

```
38 vehicle.trans_init.position  =     [10.0, 0.0, 0.0]
39 vehicle.trans_init.velocity  =     [-2.0, 0.0, 0.0]
45 vehicle.rot_init.orientation.data_source   =     trick.Orientation.InputEigenRotation
46 vehicle.rot_init.orientation.eigen_angle   =     trick.sim_services.attach_units("d",180.0)
47 vehicle.rot_init.orientation.eigen_axis    = [0.0,0.0,1.0]
48 vehicle.rot_init.ang_velocity = trick.sim_services.attach_units("d/s" , [0.0, 0.0, -36.0])
```

   NOTES:

   i. The position and velocity do not need units because they are specified in default SI units (meters, meters/s respectively). The attitude and angular rate DO need units because theya re specified in non-SI units of degrees. If they had been specified in radians, they, too, would require no units.

   ii. THIS INPUT FILE IS GETTING LARGE (MORE THAN 1 SCREEN) AND COVERS SOME SIGNIFICANT SECTIONS WITH DISPARATE VARIABLE ASSIGNMENTS. IN THE SOLUTIONS, YOU WILL SEE THAT THE MASS PROPERTIES AND VEHICLE STATE INFORMATION HAVE BEEN REMOVED TO SEPARATE FILES IN THE *Modified_data* FOLDER. THE *input* FILE ACCESSES THOSE FILES WITH THE *execfile* COMMAND.

3. If you have not yet linked to the default modules, you must do so now in order to complete the compilation. This is most effectively accomplished through the S_overrides.mk file.

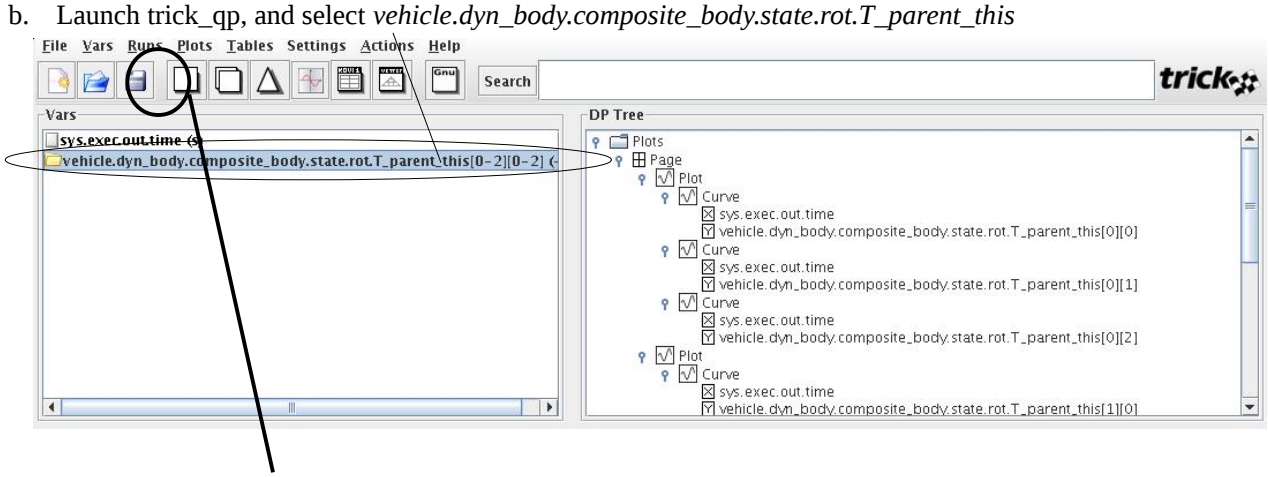
```
TRICK_CFLAGS += -I ${JEOD_HOME}/models
TRICK_CXXFLAGS += -I ${JEOD_HOME}/models
TRICK_SFLAGS += -I ${JEOD_HOME}/lib/jeod/JEOD_S_modules
```

- CP
- S_main[TAB-complete] RUN_test/input.py
  - Note – [TAB] completes the command, which will be machine specific, so we cannot tell you what the command will actually be. But [TAB] should cover all eventualities.

4. By inspection. Can be performed during the simulation using a debug (e.g. gdb) program or Trick View, or post-simulation if the data is logged. To log the data, edit the Log_data/log_state.py file (one is provided to get you started) and add the transformation matrix

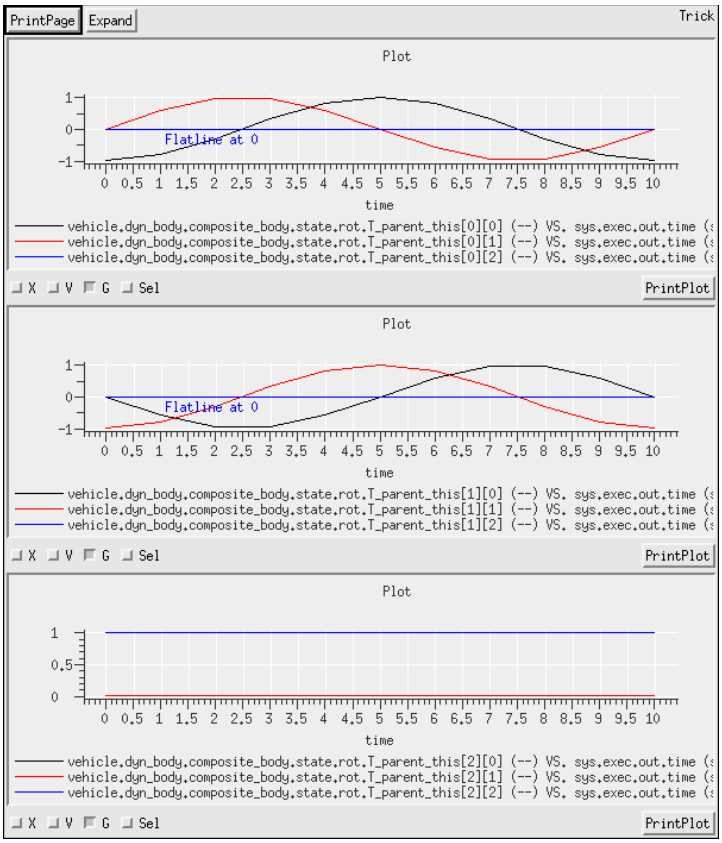
   a. `15 dr_group.add_variable( "vehicle.dyn_body.composite_body.state.rot.T_parent_this[" + str(ii) +"][" + str(jj) + "]")`

   b. Launch trick_qp, and select *vehicle.dyn_body.composite_body.state.rot.T_parent_this*



   c. Click on the plot button

   d. Look for the diagonal terms going to 1 and all other terms going to 0 at *t=5s*.


PrintPage
Expand
Trick
Plot
Flatline at 0
time
vehicle.dyn_body.composite_body.state.rot.T_parent_this[0][0] (--) VS. sys.exec.out.time
vehicle.dyn_body.composite_body.state.rot.T_parent_this[0][1] (--) VS. sys.exec.out.time
vehicle.dyn_body.composite_body.state.rot.T_parent_this[0][2] (--) VS. sys.exec.out.time
X V G Sel
PrintPlot
Plot
Flatline at 0
time
vehicle.dyn_body.composite_body.state.rot.T_parent_this[1][0] (--) VS. sys.exec.out.time
vehicle.dyn_body.composite_body.state.rot.T_parent_this[1][1] (--) VS. sys.exec.out.time
vehicle.dyn_body.composite_body.state.rot.T_parent_this[1][2] (--) VS. sys.exec.out.time
X V G Sel
PrintPlot
Plot
time
vehicle.dyn_body.composite_body.state.rot.T_parent_this[2][0] (--) VS. sys.exec.out.time
vehicle.dyn_body.composite_body.state.rot.T_parent_this[2][1] (--) VS. sys.exec.out.time
vehicle.dyn_body.composite_body.state.rot.T_parent_this[2][2] (--) VS. sys.exec.out.time
X V G Sel
PrintPlot

Exercise 6 Simple Simulation, adding clocks

1. Task, no solution
2. There is no default S_define that perfectly provides these requirements; the closest being *time_TAI_UTC_UT1_calendar.sm* which also provides UT1 clock and calendar. Including that module is the simplest solution, even if it is excessive:

```
#include “time_TAI_UTC_UT1_calendar.sm”
```

The next easiest solution is to use the jeod_time.sm S-module after indicating that UTC and clandar are needed by setting the appropriate flags:

```
#define TIME_MODEL_UTC
#define TIME_CALENDAR_UPDATE_INTERVAL 1.0
#include “jeod_time.sm”
```

Finally, the most involved way is to write your own time simulation object. This solution is provided

The sim-object can be started by copying in the basic time sim-module. The code below shows which elements come from time_dyntime_only.sm, and which are added to extend this basic sim-object:

```
##include "environment/time/include/time_manager.hh"
##include "environment/time/include/time_manager_init.hh"
##include "environment/time/include/time_tai.hh"
##include "environment/time/include/time_utc.hh"

##include "environment/time/include/time_converter_dyn_tai.hh"
##include "environment/time/include/time_converter_tai_utc.hh"

##include "environment/time/data/include/tai_to_utc.hh"

class JeodTimeSimObject : public Trick::SimObject {
  public:
   jeod::TimeManager time_manager;
   jeod::TimeManagerInit time_manager_init;

   // Time Scales
   jeod::TimeTAI time_tai;
   jeod::TimeUTC time_utc;

   // Time Converters
   jeod::TimeConverter_Dyn_TAI  conv_dyn_tai;
   jeod::TimeConverter_TAI_UTC  conv_tai_utc;

   // Instances for default data:
   jeod::TimeConverter_TAI_UTC_tai_to_utc_default_data    tai_utc_default_data;

   JeodTimeSimObject( ){

      //
      // Default data jobs
      //

      ("default_data") tai_utc_default_data.initialize( &conv_tai_utc );

      //
      // Initialization jobs
      //

      // TAI
      P_TIME ("initialization") time_manager.register_type(time_tai);
      P_TIME ("initialization") time_manager.register_converter(conv_dyn_tai);
```

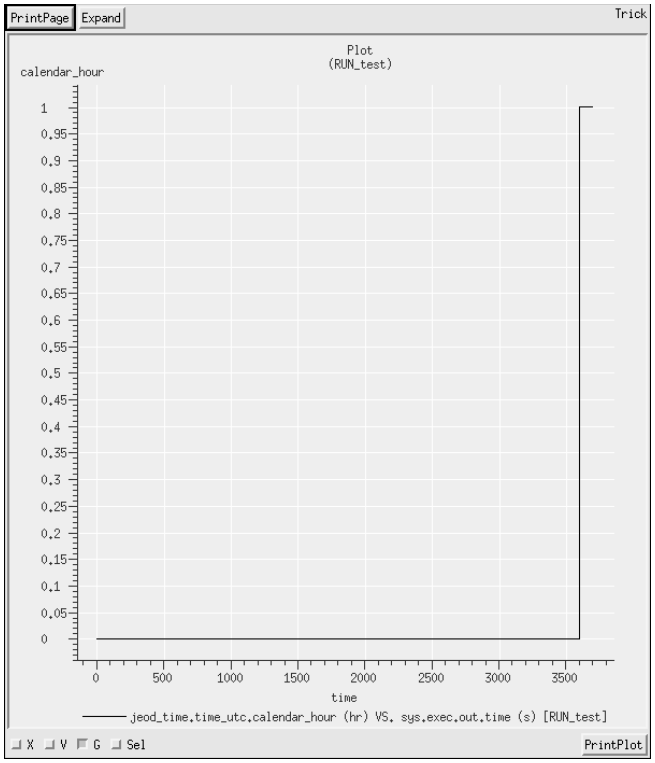
```
      // UTC
      P_TIME ("initialization") time_manager.register_type(time_utc );
      P_TIME ("initialization") time_manager.register_converter(conv_tai_utc );

      // Initialize the time manager.
      P_TIME ("initialization")  time_manager.initialize(&time_manager_init);

      // Compute appropriate calendar dates at initialization.
      P_TIME ("initialization") time_utc.calendar_update(exec_get_sim_time() );

      //
      // Scheduled jobs
      //
      // Update Time Scales
      (DYNAMICS, "environment") time_manager.update( exec_get_sim_time() );

      // Update the calendar times of interest.
      (DYNAMICS, "environment") time_utc.calendar_update( exec_get_sim_time() );
   };

  private:
   // This object is not copyable
   JeodTimeSimObject (const JeodTimeSimObject&);
   JeodTimeSimObject & operator = (const JeodTimeSimObject&);

};
JeodTimeSimObject jeod_time;
```

3. The solution sets this to 3700 seconds, any value > 3600 seconds is good.

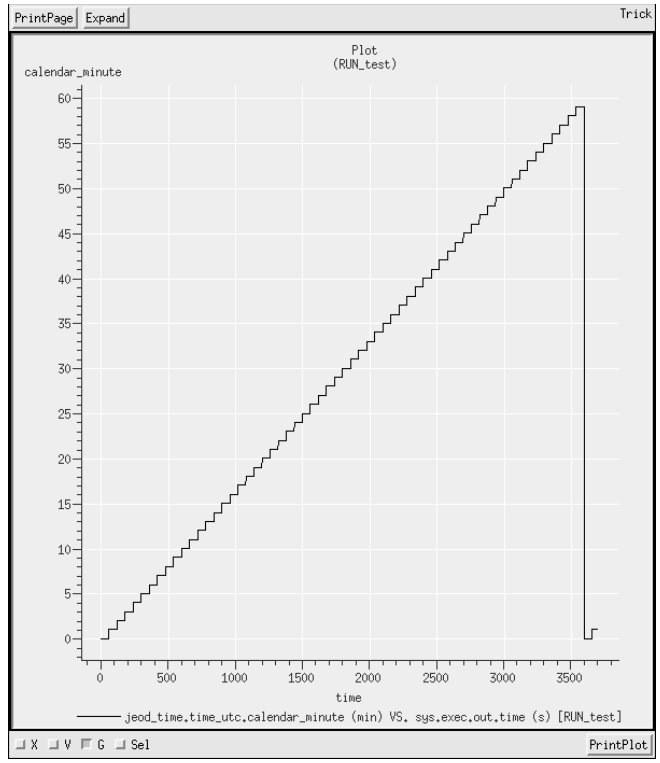
```
32 trick.sim_services.exec_set_terminate_time(3700.0);
```

4. The time settings are put in a new Modified_data file, *time.py.*

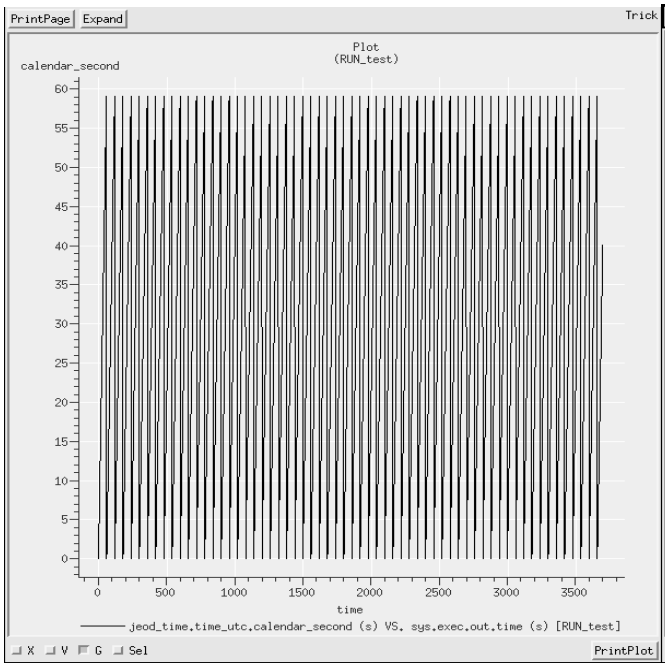
5. Two options here, either add the new values to the existing *log_state.py* file, or – since time does not really belong in a file named *log_state.py* – add a new file with a more appropriate name, such as *log_time.py.* The solutions illustrate the latter option. Here we log seconds as well as calendar_seconds to illustrate the difference. *seconds* represents the number of seconds elapsed since epoch (J2000), while *calendar_seconds* cycles from 0 to 60 and returns.



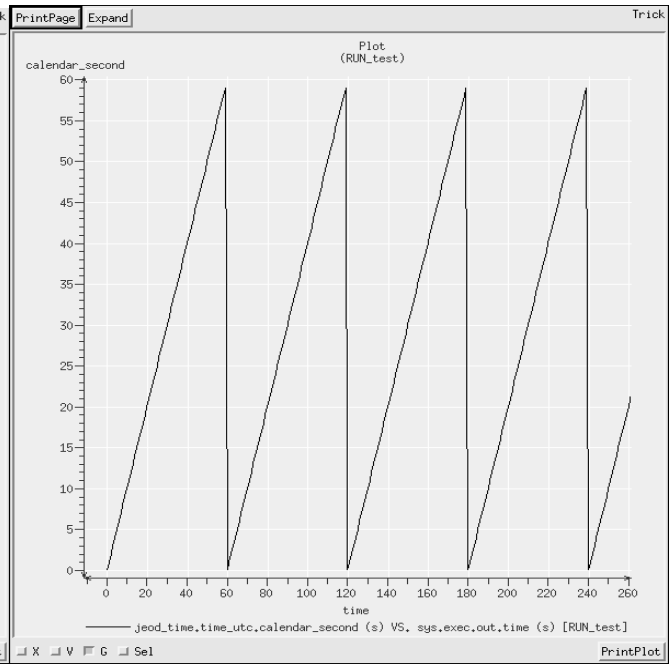
Hours



Minutes

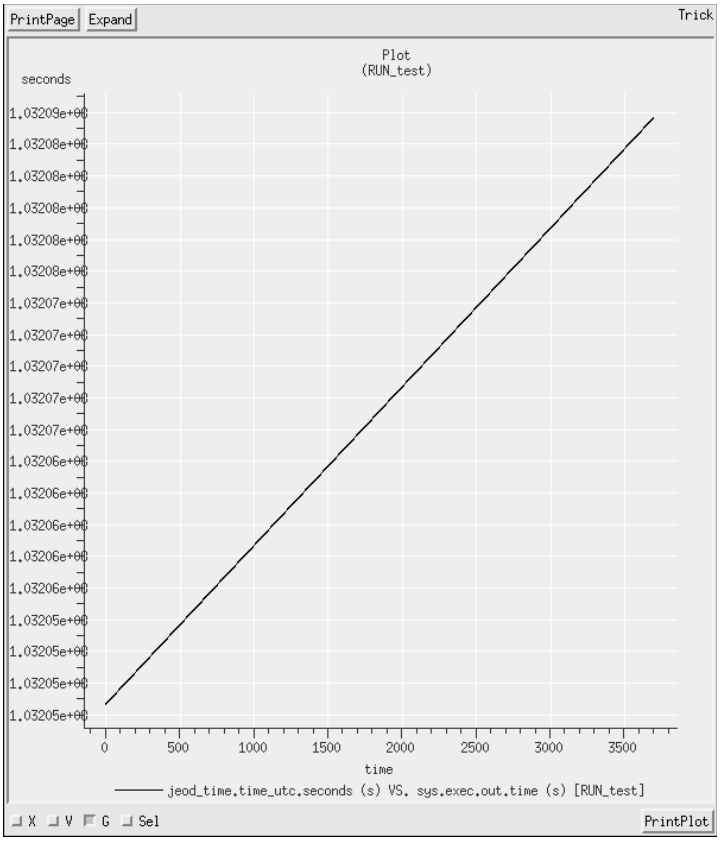


calendar-seconds



calendar-seconds, zoomed in

Value climbs to 59.0 before resetting to 0.0. Since data is logged every second, cannot get higher than 59.0 (60.0 is illegal, it is equivalent to 0.0).



seconds since epoch, increases monotonically

Exercise 7 Two-vehicle Simulation

1. Task, no solution
2. Changing the S_define involves only instantiating another instance of the VehichleSimObject, copying the call directly from the *vehicle_basic.sm* file.

```
26 VehicleSimObject vehicle2(dynamics.dyn_manager);
```

Changing the input file means adding new lines defining the vehicle behavior:

```
23 vehicle2.dyn_body.set_name("veh2")
24 vehicle2.dyn_body.integ_frame_name = "Space.inertial"
25 vehicle2.dyn_body.translational_dynamics = True
26 vehicle2.dyn_body.rotational_dynamics = True
```

new Modified data files defining the vehicle properties and state:

```
30 execfile("Modified_data/vehicle2_mass_props.py")
31 execfile("Modified_data/vehicle2_state.py")
```

and adding the new BodyAction items to the dynamics manager

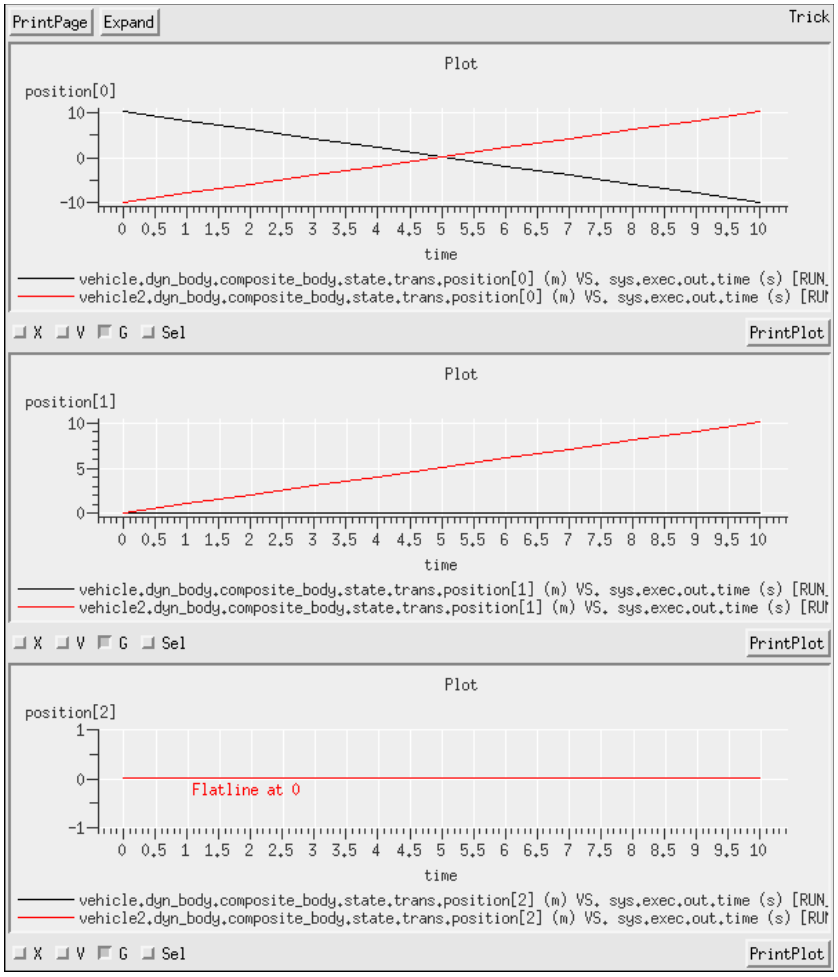
```
38 dynamics.dyn_manager.add_body_action( vehicle2.mass_init )
39 dynamics.dyn_manager.add_body_action( vehicle2.trans_init )
40 dynamics.dyn_manager.add_body_action( vehicle2.rot_init )
```

3. The Modified data files were generated by copying the old files to vehicle2_*, and changing all instances of *vehicle* to *vehicle2* in them. The inertia specification can be changed from the default body-axes to the structural axes – as described in section 4.1.1 of the Mass Model documentation – with the instruction:

```
4 vehicle2.mass_init.properties.inertia_spec =trick.MassPropertiesInit.Struct
```

4. Task, no solution
5. There are at least three ways to do this:
   (a) Add both vehicles to one log file (method illustrated in solutions)
   (b) Make a second log file as we did for SIM_06, this time for the state of the second vehicle; perhaps name it veh2_state.py.
   (c) Make a generic log file, use the character string equal to the vehicle identification as an argument (i.e. “vehicle” or “vehicle2”, and call the method for each vehicle (method illustrated in *verif/SIM_dyncomp_T10/Log_data/log_sv_state_rec.py*)

   The output graphs will depend on the initial conditions. In the solutions, the x-axis motion of vehicle2 mirrors that of vehicle1; vehicle2 also has a velocity on the y-axis and their z-components are identical.


PrintPage
Expand
Trick
Plot
position[0]
10
0
-10
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.5 9 9.5 10
time
vehicle.dyn_body.composite_body.state.trans.position[0] (m) VS. sys.exec.out.time (s) [RUN
vehicle2.dyn_body.composite_body.state.trans.position[0] (m) VS. sys.exec.out.time (s) [RU
X V G Sel
PrintPlot
Plot
position[1]
10
5
0
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.5 9 9.5 10
time
vehicle.dyn_body.composite_body.state.trans.position[1] (m) VS. sys.exec.out.time (s) [RUN
vehicle2.dyn_body.composite_body.state.trans.position[1] (m) VS. sys.exec.out.time (s) [RU
X V G Sel
PrintPlot
Plot
position[2]
1
0
-1
Flatline at 0
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 7.5 8 8.5 9 9.5 10
time
vehicle.dyn_body.composite_body.state.trans.position[2] (m) VS. sys.exec.out.time (s) [RUN
vehicle2.dyn_body.composite_body.state.trans.position[2] (m) VS. sys.exec.out.time (s) [RU
X V G Sel
PrintPlot

Exercise 8 Creating a Simple Spherical-Earth Orbiter

1. Task, no solution
2. Task, no solution
3. The simplest way to do this is to add the basic environment default module, *environment_sans_de405.sm*. There is no need for the Ephemerides, so *environment.sm* is excessive (indeed, because Terrestrial Time is not defined in the chosen time S-module, *environment.sm* will not work).

```
20 #include “environment_sans_de405.sm”
```

4. The simplest way to do this is to add the basic earth-representation default module, *earth_basic.sm.*

```
21 #include “earth_basic.sm”
```

5. We should be working in SinglePlanet mode because we have a single planet in the simulation. The *central_point_name* must be corrected to indicate that we have *Earth* in the simulation.

```
11 dynamics.dyn_manager_init.mode =         trick.DynManagerInit.EphemerisMode_SinglePlanet
12 dynamics.dyn_manager_init.central_point_name = "Earth"
```

6. Added the file *vehicle_grav_control.py*. Included it with:

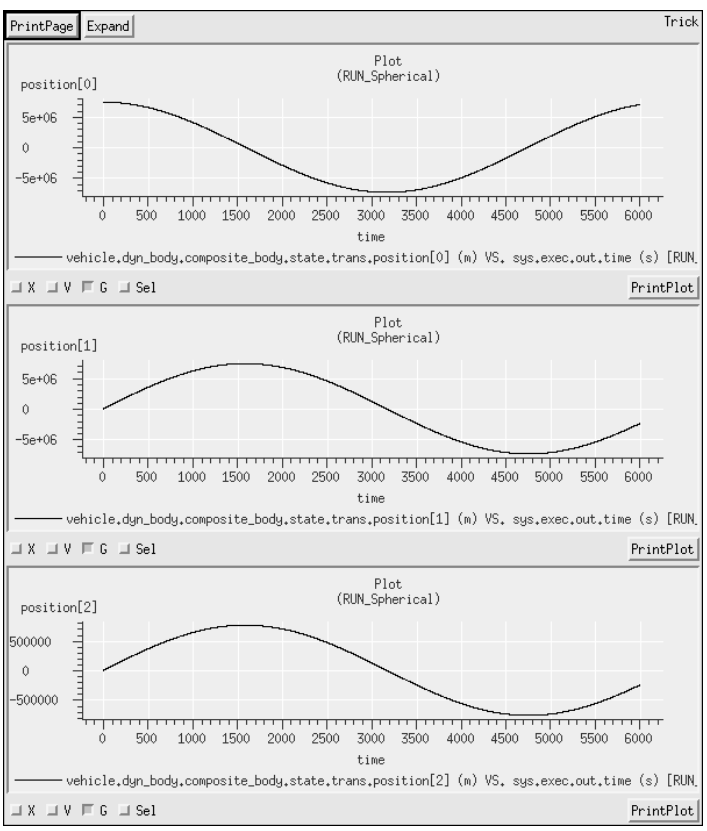
```
25 execfile("Modified_data/vehicle_grav_controls.py")
```

7. See the file *Modified_data/vehicle_grav_control.py*
8. Changed the former vehicle-state initialization call to

```
24 execfile("Modified_data/vehicle_state_ex8.py")
```

9. Changed the simulation stop time:

```
32 trick.sim_services.exec_set_terminate_time(6000.0);
```



Exercise 9 Creating a Simulation with Two Earths

1. Task
2. A simple change to the S_define:

```
 23 EarthSphericalSimObject earth2( dynamics.dyn_manager, env.gravity_manager);
```

3. The run faults with a fatal error:

```
DIAGNOSTIC: Fatal Error environment/gravity/duplicate_entry:
Duplicate planet name 'Earth'
```

   It is not possible to accidentally create two identical planets and have your vehicle around the wrong one.

4. Whether the attempt is made to change *earth2.planet.name*, or *earth2.gravity_body.planet_name*, there is still a fatal error. You may see 2 errors:

```
Error environment/ephemerides/duplicate_entry at ephem_manager.cc line 133:
Planet with name 'Earth' was previously registered.
```

```
Error utils/ref_frames/duplicate_entry at ref_frame_manager.cc line 94:
Reference frame with name 'Earth.inertial' was previously registered.
```

   before failing with a fatal error:

```
DIAGNOSTIC: Fatal Error environment/ephemerides/invalid_item:
Illegal attempt to change the target frame for ephemeris item 'Earth.inertial'.
```

Exercise 10 Complications of Adding a General Gravity Model

1. Task
2. There is a default S_module that contains the GGM05C field already configured. Include that in place of earth_basic.sm

```
S_define
31 #include "earth_GGM05C.sm"
```

3. Task.

Exercise 11 Creating a Spherical-Earth Orbiter using a Non-spherical Gravity model

1. task
2. Easiest solution is to use the pre-made simulation object:

```
S_define
19 #include "time_TAI_UTC_UT1_TT_GMST.sm"
```

3. #define the rate LOW_RATE_ENV at the top of the S_define:

```
S_define
 8 #define LOW_RATE_ENV  100.0
```

4. Task
5.
   (a) Adding earth to the integration list.

   In the default module, the axial rotation calculation is specified as a derivative-class job, Therefore its class (earth) must be added to the list of objects in the integration list.

```
S_define
```

```
 26 IntegLoop sim_integ_loop (DYNAMICS) dynamics, earth;
```

(b) Setting the absolute time.

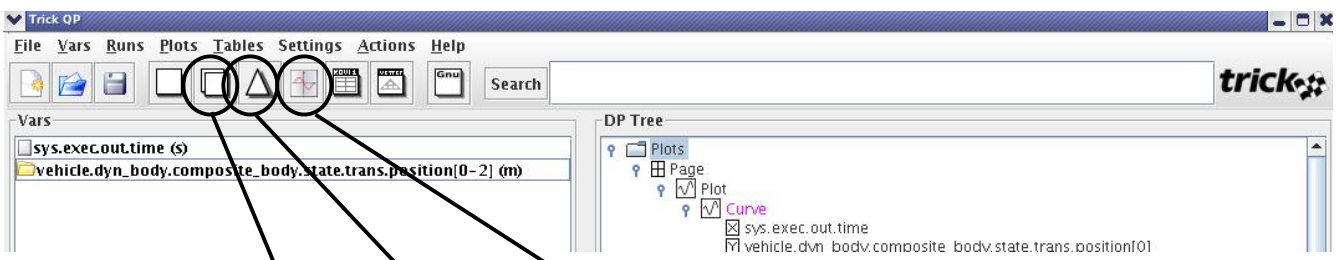
Since we have added absolute clocks, it is also necessary to set the simulation in an absolute time. For this, the Modified data file from SIM_06 was copied in and extended to accommodate the additional clocks. Because UTC was left in the time module, there was no need to re-specify the initialization time.

```
Modified_data/time.py
1-18 copied from SIM_06/Modified_data/time.py
 19 jeod_time.time_ut1.initialize_from_name = "TAI"
 20 jeod_time.time_ut1.update_from_name = "TAI"
 21
 22 jeod_time.time_tt.initialize_from_name = "TAI"
 23 jeod_time.time_tt.update_from_name = "TAI"
 24
 25 jeod_time.time_gmst.initialize_from_name = "UT1"
 26 jeod_time.time_gmst.update_from_name = "UT1"
```

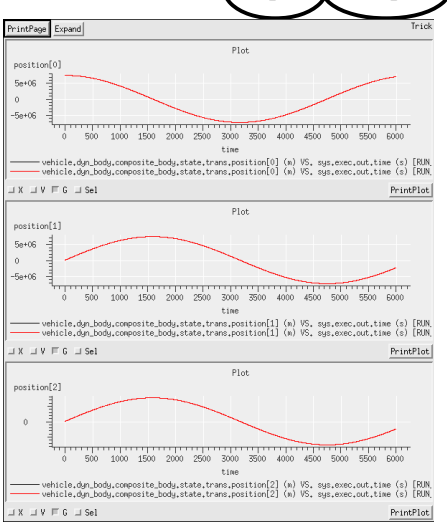
The gravity controls file from the SIM_08 solution should still be available, but if not, here it is again.

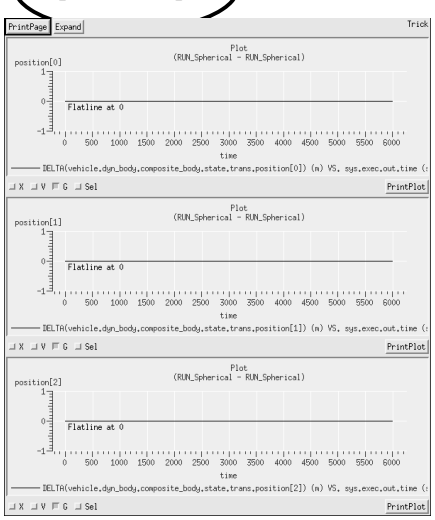
```
Modified_data/vehicle_grav_controls.py
  1 vehicle.grav_control.source_name = "Earth"
  2 vehicle.grav_control.active = True
  3 vehicle.grav_control.spherical = True
  4
  5 vehicle.dyn_body.grav_interaction.add_control(vehicle.grav_control)
```

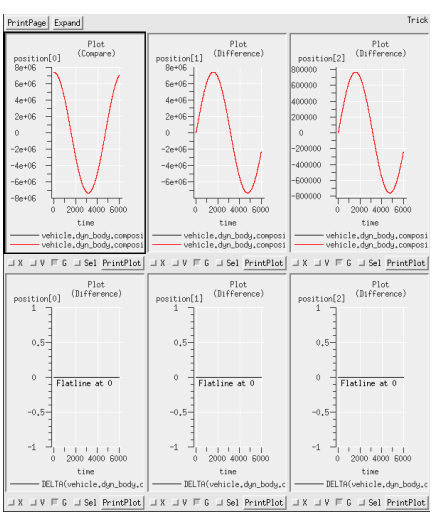
6. Using trick_qp:



click on the co-plot, delta-plot, or co-plot-delta-plot combination button and get, respectively:







for the position data.

Note that when comparing two sets of close data, the comparison plot tool may not be useful because the width of the line can obliterate any data differences. The delta tool is more useful in these cases.

In this case, the delta-plot tool shows no difference between the runs. Running with the spherical gravity model and running with the spherical option on the GGM05C gravity model are equivalent (as we would hope). The only difference is that the GGM05C model requires all of the additional baggage assocaited with the RNP model. If the simualtion requires spherical gravity, it is preferable to use a spherical-gravity S-module rather than configuring the general S-module to operate with spherical gravity.

Exercise 12 Creating a Non-spherical-Earth Orbiter

1. task
2. Made a new Modified_data file *vehicle_grav_controls_non_spherical.py*, and moved the generic file to *vehicle_grav_controls_spherical.py* to distinguish the purpose of the two files more clearly.

   The non-spherical Modified-data file defines a function, allowing the degree and order to be set from the input file.

Modified_data/vehicle_grav_controls_non_spherical.py

```
def veh_grav_controls(degree,order) :
  vehicle.grav_control.source_name = "Earth"
  vehicle.grav_control.active = True
  vehicle.grav_control.spherical = False
  vehicle.grav_control.degree = degree
  vehicle.grav_control.order = order

  vehicle.dyn_body.grav_interaction.add_control(vehicle.grav_control)

  return
```

   NOTE – spherical gravity can also be achieved by setting the degree and order to 0 in a non-spherical model. Thus, the old Modified_data is actually redundant, but is kept here for continuity. It will be abandoned a little later.

3. Change the execfile command, and call the new function with degree = 4, order = 4:

RUN_4x4/input.py

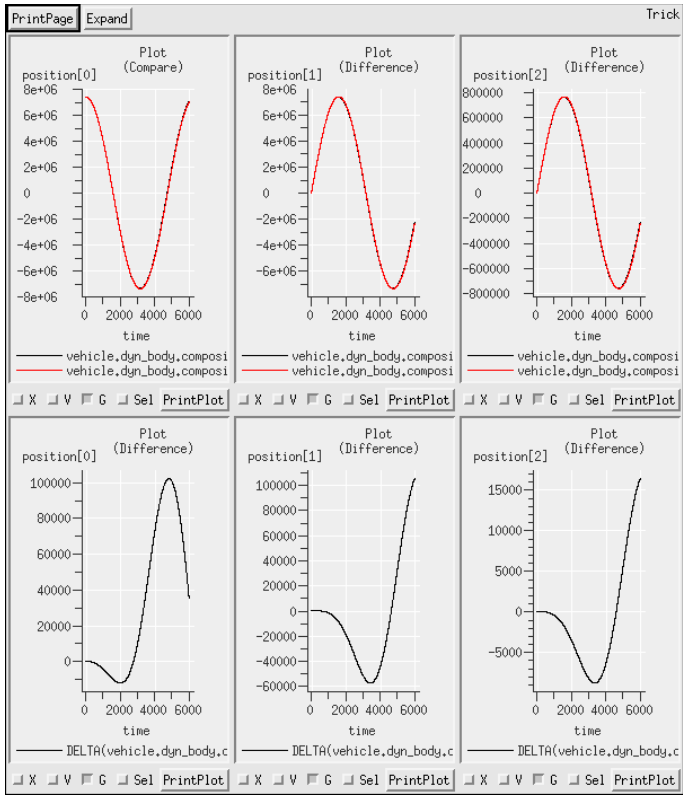
```
execfile("Modified_data/vehicle_grav_controls_non_spherical.py")
veh_grav_controls(4,4)
```

4. Similar to the 4x4 run

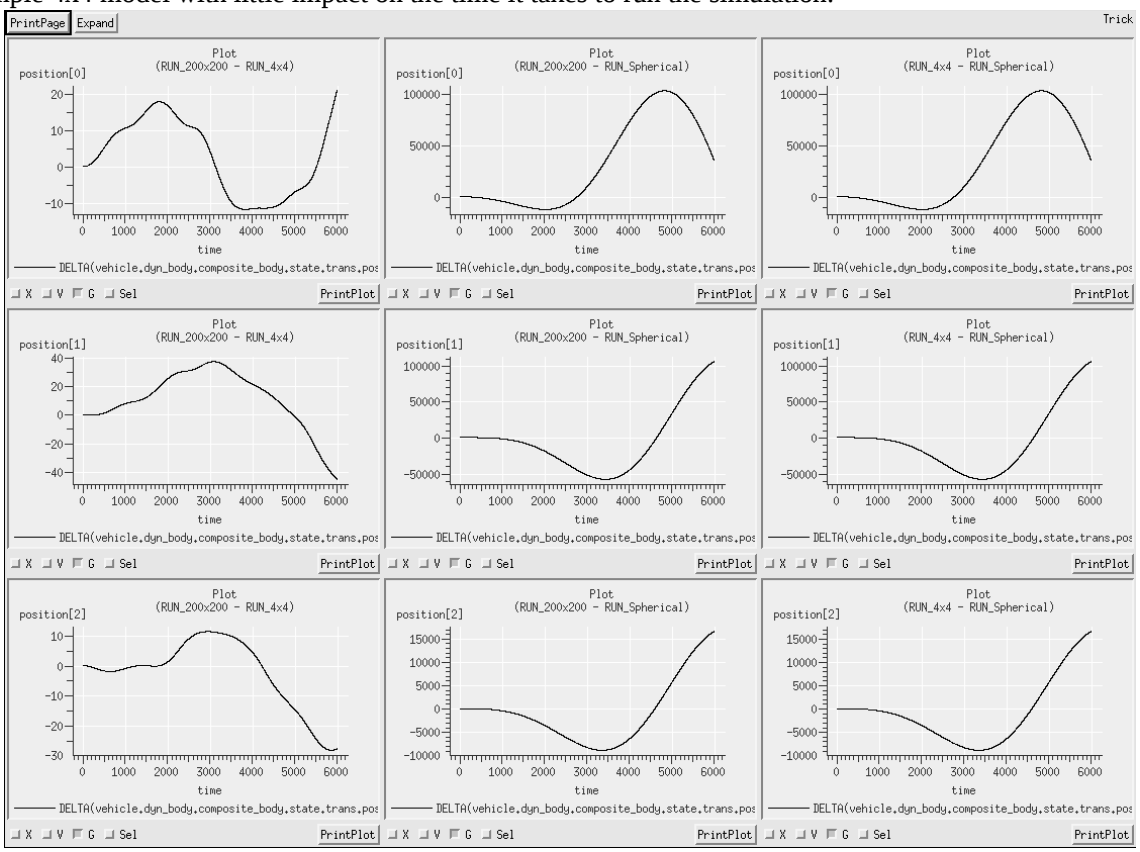
RUN_200x200/input.py

```
execfile("Modified_data/vehicle_grav_controls_non_spherical.py")
veh_grav_controls(200,200)
```

5. Position comparison and delta plot shown, first for the simple comparison of 4x4-vs-Spherical, then for all 3.



Note that this is an illustrative example of how the difference on the co-plot is obliterated by the line width, while difference is quite visible on the delta-plot. The difference is significant, 100km in position between the spherical and the simple 4x4 model with little impact on the time it takes to run the simulation.



This shows the position deltas between the different run combinations. Note that the difference between the 4x4 and 200x200 model changes the position data by less than 100m, while it takes 60-70 times longer to run with a 200x200 gravity model than with a 4x4.

The lesson here is that the low-fidelity terms in the gravity model are quick and easy to compute and have significant effect on the vehicle, while the high-fidelity terms are progressively more expensive to compute and have progressively less impact on the trajectory. There is a trade-off between simulation speed and fidelity, and the optimum configuration is often fairly low fidelity, 4x4, 8x8 or the like. Low-altitude simulations are more likely to benefit from the additional expense of higher fidelity models, but for earth-based simulations, uncertainties in atmospheric conditions and the resulting aerodynamic drag can quickly exceed the effects of high-fidelity gravity at low altitude.

Exercise 13 Multiple Gravitational Fields

1. task
2. Since we only need spherical gravity the moon_basic and sun_basic default modules suffice.

```
S_define
 23 #include "moon_basic.sm"
 24 #include "sun_basic.sm"
```

3. There are two environment sim modules, we have been using the one without ephemerides up to this point. Change it.

```
S_define
 21 #include "environment.sm"
```

   (a) In the input file, change mode to *Ephemerides*

```
RUN_Spherical/input.py
 11 dynamics.dyn_manager_init.mode
        =trick.DynManagerInit.EphemerisMode_Ephemerides
```

   (b) Need Terrestrial Time, but already have that, and absolute time configuration from SIM_12

   (c) Because we are now using the Ephemerides Model, we need to add to the S_overrides.mk file in the SIM-directory at the same level as the S_define.

```
 8 all: build_ephem_files
 9
10 EPHEM_MODEL = 405
11 build_ephem_files:
12     @ echo "Building JPL DE"${EPHEM_MODEL} " ephemeris files" ;\
13     cd ${JEOD_HOME}/models/environment/ephemerides/de4xx_ephem/data ;\
14     ${MAKE} MODEL=${EPHEM_MODEL} all
```

4. There is not a standard S-module for a vehicle with 3 gravity controls, so there are two options to replace the vehicle_basic sim_module:

   (a) enter the entire module, and add the lines for the gravity controls, replacing the one existing line with three planet-specific instances. This solution is presented for this exercise.

   (b) inherit the Base S-module and add the three planet-specific gravity controls as the only new elements. This solution is presented in Exercise 14.

```
S_define
 46    jeod::SphericalHarmonicsGravityControls  earth_grav_control;
 47    jeod::SphericalHarmonicsGravityControls  moon_grav_control;
 48    jeod::SphericalHarmonicsGravityControls  sun_grav_control;
```

   (c) For the input files, the Modified_data/*vehicle_grav_controls_non_spherical.py* was moved back to *vehicle_grav_controls.py* and spherical gravity will be specified by setting the degree and order to be zero.

5. Task
6. Edited the gravity controls file to take two additional arguments indicating whether lunar and solar gravity are on.

```
Modified_data/vehicle_grav_controls.py
1 def veh_grav_controls(degree, order, moon_active=False, sun_active=False) :
```

(a) Earth and Sun:

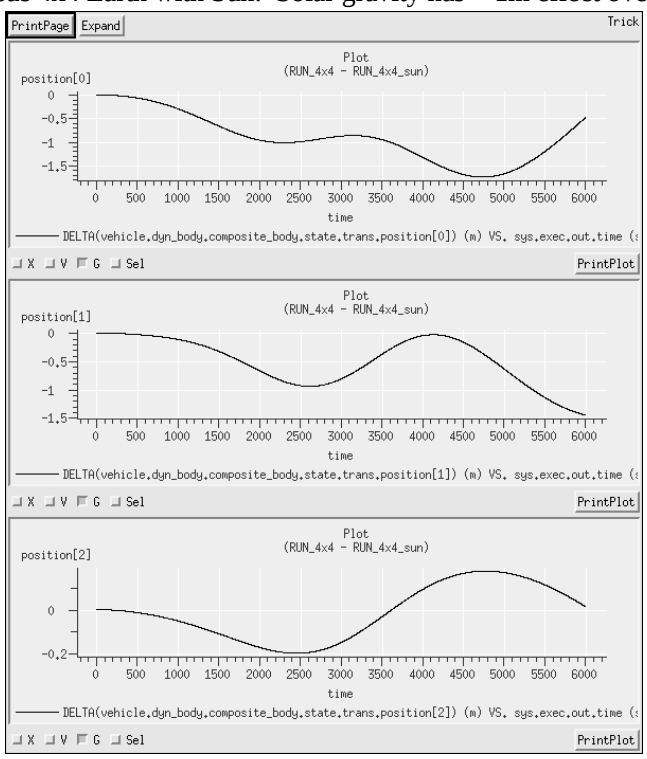
```
RUN_4x4_sun/input.py
 27 veh_grav_controls(4,4, sun_active=True)
```

(b) adding interactions:

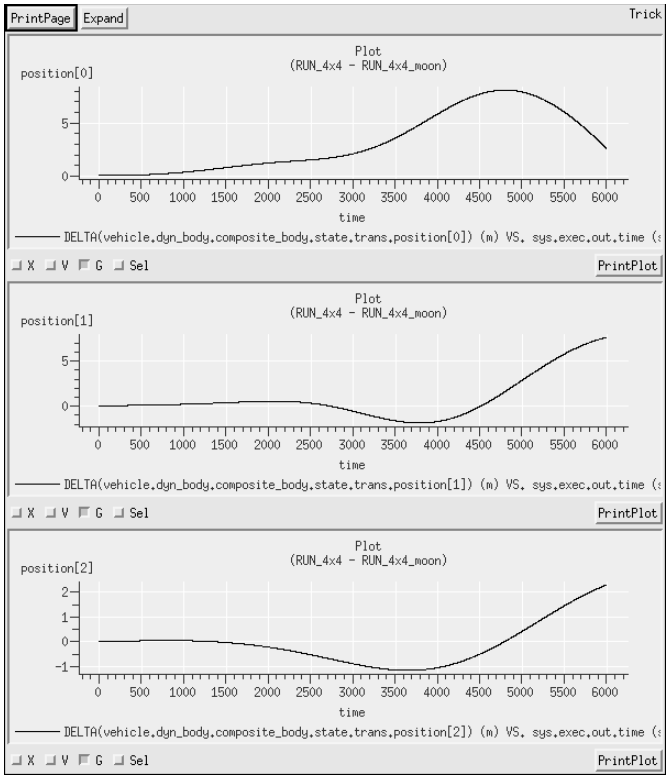
```
Modified_data/vehicle_grav_controls.py
17 vehicle.dyn_body.grav_interaction.add_control(vehicle.earth_grav_control)
18 vehicle.dyn_body.grav_interaction.add_control(vehicle.moon_grav_control)
19 vehicle.dyn_body.grav_interaction.add_control(vehicle.sun_grav_control)
```

7.

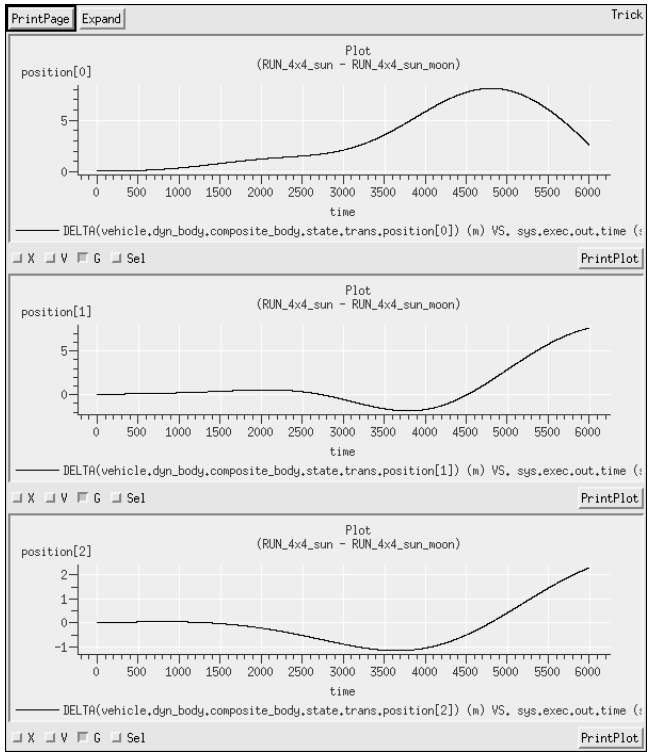
(a) 4x4 Earth alone versus 4x4 Earth with Sun. Solar gravity has ~ 1m effect over 1 orbit.



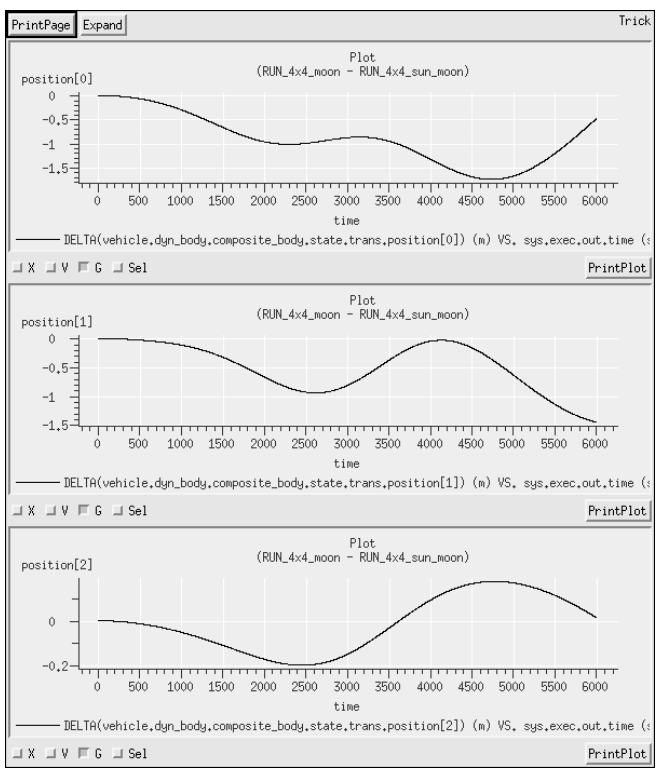
(b) 4x4 Earth alone versus 4x4 Earth with Moon. Lunar gravity has ~ 10m effect over 1 orbit.



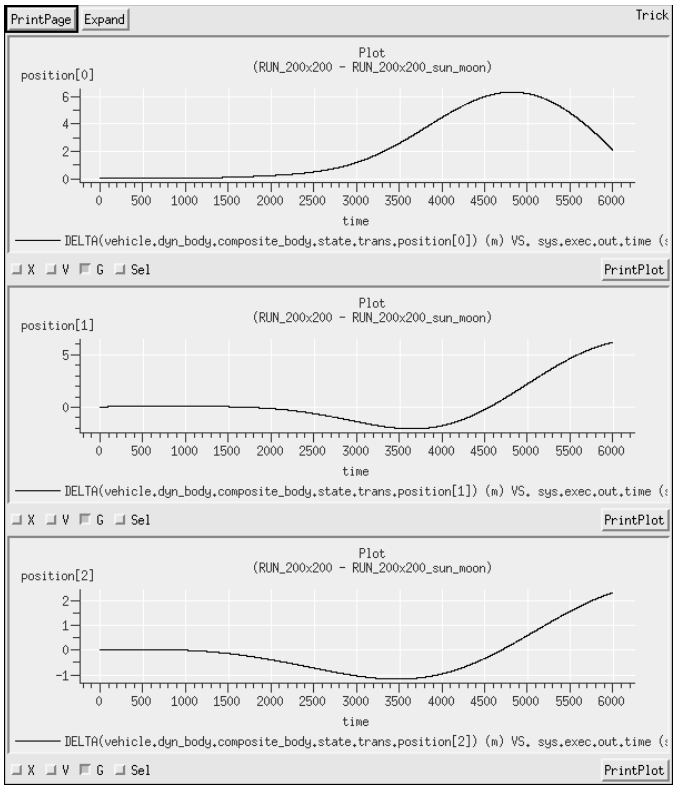
(c) 4x4 Earth with Sun and Moon versus 4x4 Earth with Sun only. Lunar gravity shows the same ~10m effect, as should be expected.



(d) 4x4 Earth with Sun and Moon versus 4x4 Earth with Moon only. Solar gravity shows the same ~1m effect, as should be expected.



(e) 200x200 Earth alone versus 200x200 Earth with Sun and Moon. Perturbation effects are of the same order as for the 4x4 case.



Exercise 14 Lunar Orbiting Vehicles

1. Task
2. Integrating state, so need dynamics.sm

   Using multiple planets, so need Ephemeris Model, so need environment.sm

   Using all spherical gravity, so use earth_basic.sm, moon_basic.sm, and sun_basic.sm

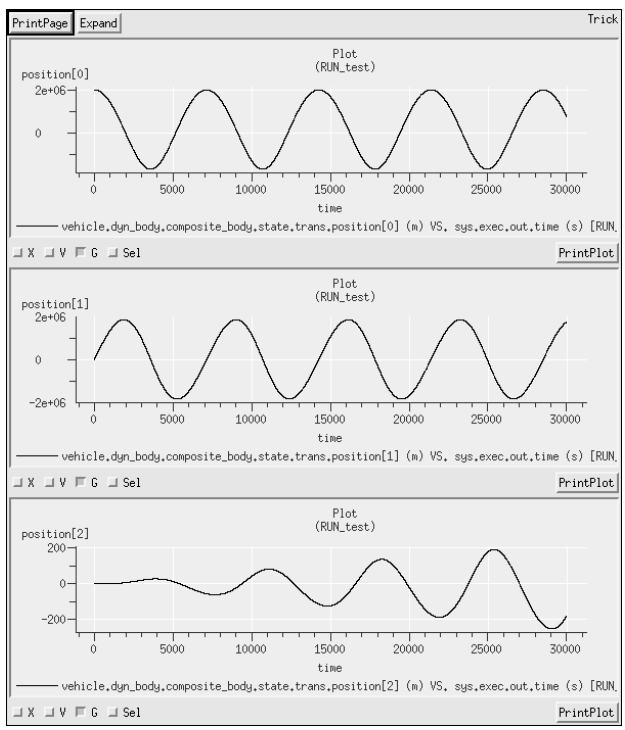
   Need a 3-controls vehicle, so added these controls to an inherited baseline of the basic vehicle simualtion-object.

```
S_define
 33   jeod::SphericalHarmonicsGravityControls  earth_grav_control;
 34   jeod::SphericalHarmonicsGravityControls  moon_grav_control;
 35   jeod::SphericalHarmonicsGravityControls  sun_grav_control;
```

   Using Ephemerides, so need Terrestrial Time. Using the redundant time_TAI_UTC_UT1_TT.sm to avoid having to write a new time module.

   Looking at planetary and gravity data for Moon (models/environment/planet/data and models/environment/gravity/data), we have a radius of 1738km, so set the initial position somewhat larger (2000km). Have a gravity constant $\mu = 4.9 \times 10^{12} \; kg \cdot m^3 \cdot s^{-2}$, so a circular orbital velocity of approximately $1500 \, m \; s^{-1}$.

```
Modified_data/vehicle_state.py
  2 vehicle.trans_init.reference_ref_frame_name = "Moon.inertial"
  3 vehicle.trans_init.body_frame_id            = "composite_body"
  4 vehicle.trans_init.position                 = [2000000.0,    0.0, 0.0]
  5 vehicle.trans_init.velocity                 = [      0.0, 1500.0, 0.0]
```



Exercise 15 Gravity Gradient

1. Task

2. *vehicle_basic.sm* does not have gravity gradient, so included this class verbosely, and added the following lines:

```
S_define
 28 ##include "interactions/gravity_torque/include/gravity_torque.hh"
...
 34    jeod::GravityTorque                      grav_torque;
...
 40       P_GRAV ("initialization") grav_torque.initialize(dyn_body);
 41
 42       P_PGRAV ("derivative") grav_torque.update();
...
 50 vcollect vehicle.dyn_body.collect.collect_no_xmit_torq jeod::CollectTorque::create {
 51    vehicle.grav_torque.torque};
```

   Because the gradient torque was made a derivative job within the vehicle class, the vehicle class needs adding to the integration statement:

```
 54 IntegLoop sim_integ_loop (DYNAMICS) dynamics, vehicle;
```

3. Only one line is needed because there is no specification for degree or order (and there can be no non-redundant, non-zero specification, since this is a spherical gravity field).

```
Modified_data/vehicle_grav_controls.py
  4 vehicle.grav_control.gradient = True
```

4. Made two runs, one for each of the new mass-properties files. Called on RUN_regular_body and the other RUN_irregular_body. Other than the mass-properties definitions, the rest of the input file is the same, so there are two choices, both equally valid:

   (a) In each of the new runs, execfile the input file from RUN_Spherical (copied in from earlier simulation), then overwrite the mass properties by running execfile on the appropriate new Modified-data file. The new input files would comprise only two lines, and look like:

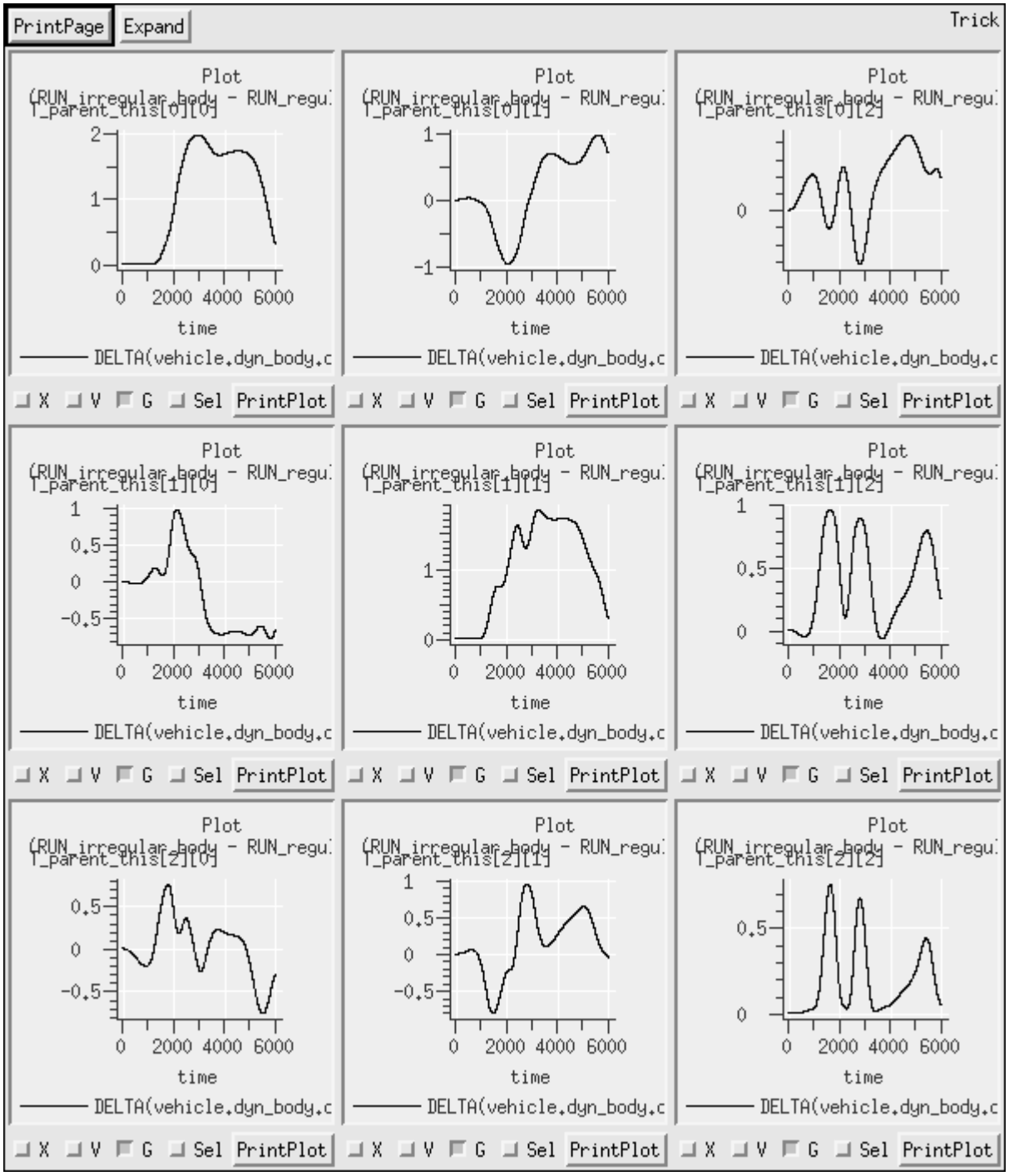
```
execfile(“RUN_Spherical/input.py”)
execfile(“Modified_data/vehicle_mass_props_regular_body.py”)
```

   (b) Copy the input file from RUN_Spherical into each of the new RUN-directories, and edit the execfile command to access the appropriate mass-properties file.

```
RUN_regular_body/input.py
23 execfile("Modified_data/vehicle_mass_props_irregular_body.py")
```

   Under option (a), the old files must be kept around, under option (b) they can be deleted. Since the input file is not very large, duplicating it is not very cumbersome, so option (b) was selected.

5. The position data for the vehicles should show no difference between the two runs. The orientation data (here we plotted T_parent_this) shows that one vehicle (regular-body) holds steady, while the other (irregular-body) rotates on all three axes. Plot below shows the differences between the two transformation matrices



Exercise 16 Solid Body Tides

1. task
2. The Earth simulation-object needs the new model adding. The vehicle simulation-object needs to be expanded from the default simulation-modules, and edited. I'm going to use the default data file, so must include Moon and Sun

   (a) For the Earth simulation object, the following lines are added:

```
31 ##include "environment/gravity/include/spherical_harmonics_solid_body_tides.hh"
32 ##include  "environment/gravity/include/spherical_harmonics_solid_body_tides_init.hh"
33 ##include "environment/gravity/data/include/earth_solid_tides.hh"
...
39   jeod::SphericalHarmonicsSolidBodyTidesInit sbtide_init;
40   jeod::SphericalHarmonicsSolidBodyTides sb_tide;
...
42   jeod::SphericalHarmonicsSolidBodyTidesInit_earth_solid_tides_default_data    sbtide_init_def ault_data;
...
56      ("default_data") sbtide_init_default_data.initialize ( &sbtide_init );
...
61      P_EPH  ("initialization") gravity_source.add_deltacoeff (
62         sbtide_init,
63         dyn_manager,
64         sb_tide );
```

   (b) For the vehicle simulation object, the following lines were added:

```
86 ##include "environment/gravity/include/spherical_harmonics_delta_controls.hh"
...
92   jeod::SphericalHarmonicsDeltaControls sbtide_ctrl;
```

   (c) Add simple implementations of Moon and Sun and change the environment from environment_sans_de405.sm to environment.sm to bring in the ephemerides of Sun and Moon:

```
78 #include "moon_basic.sm"
79 #include "sun_basic.sm"
```

3. Using the same input-file call to veh_grav_controls as from SIM_12:

```
27 veh_grav_controls(8,8)
```

4. Made two directories – RUN_tides_on, and RUN_tides_off. Edited the gravity setup Modified-data file (*vehicle_grav_controls.py*) to add the solid-body-tides information:

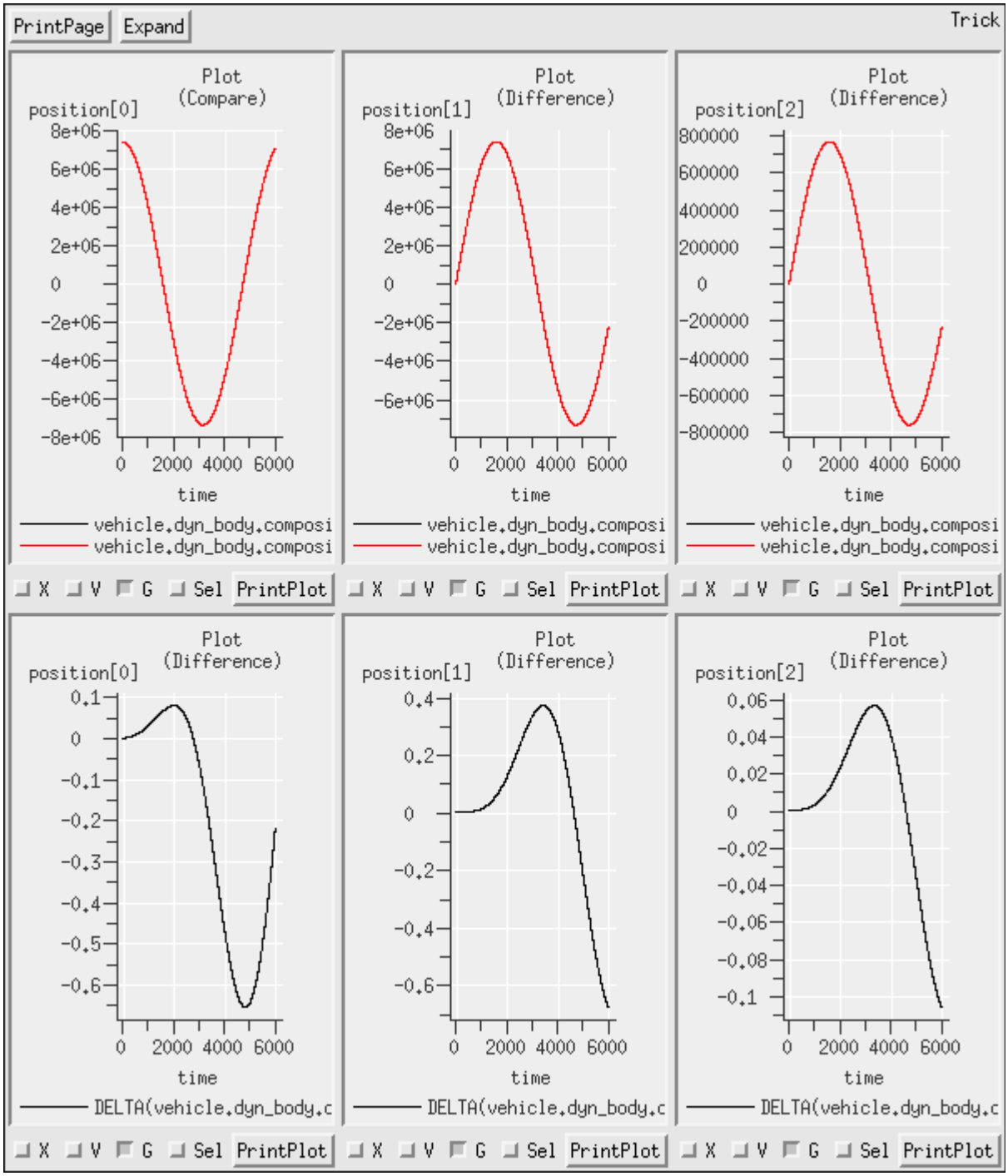
```
 8   vehicle.sbtide_ctrl.first_order_only = True
 9   vehicle.sbtide_ctrl.grav_effect = earth.sb_tide
10   vehicle.sbtide_ctrl.grav_source   = earth.gravity_source
11   vehicle.sbtide_ctrl.active = True
```

   Here I default the solid-body-tides to be on, but turn them off again in the RUN_tides_off input file:

```
28 vehicle.sbtide_ctrl.active = False
```

   Alternatively, I could have passed this argument into the veh_grav_controls function, along with the degree and order., so line 27 would be, e.g. *veh_grav_controls(8,8,True)*

5. Position data is shown for approximately one orbit.


PrintPage
Expand
Trick
Plot
(Compare)
position[0]
8e+06
6e+06
4e+06
2e+06
0
-2e+06
-4e+06
-6e+06
-8e+06
0 2000 4000 6000
time
vehicle.dyn_body.composi
vehicle.dyn_body.composi
X V G Sel PrintPlot
Plot
(Difference)
position[1]
8e+06
6e+06
4e+06
2e+06
0
-2e+06
-4e+06
-6e+06
0 2000 4000 6000
time
vehicle.dyn_body.composi
vehicle.dyn_body.composi
X V G Sel PrintPlot
Plot
(Difference)
position[2]
800000
600000
400000
200000
0
-200000
-400000
-600000
-800000
0 2000 4000 6000
time
vehicle.dyn_body.composi
vehicle.dyn_body.composi
X V G Sel PrintPlot
Plot
(Difference)
position[0]
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
0 2000 4000 6000
time
DELTA(vehicle.dyn_body.c
X V G Sel PrintPlot
Plot
(Difference)
position[1]
0.4
0.2
0
-0.2
-0.4
-0.6
0 2000 4000 6000
time
DELTA(vehicle.dyn_body.c
X V G Sel PrintPlot
Plot
(Difference)
position[2]
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
-0.1
0 2000 4000 6000
time
DELTA(vehicle.dyn_body.c
X V G Sel PrintPlot

Exercise 17 Interpretation of Variables

1. The first one specifies that the center of mass is 1.0 m along the x-axis from the origin of the integration frame. The x-axis in this case is that of the integration frame.

   The second specifies that the center of mass is at -1.0m along the structural-frame x-axis, measured from the origin of the structural-frame. The structural-frame is that of *d_body.*

   The third specifies that the center of mass is at -1.0m along the x-axis of the structure-point partial frame of object *m_body,* measured from the structure-point.

2. No, there is no information about the relative location of the structure-point of m_body relative to either the integration frame or the structural-frame of *d_body.*

3. No, there is also no information on the relative orientations.

Exercise 18 Adding Mass Points and Internal Reference Frames

1. Task
2. Mass Point added using the MassBodyInit Body Action:

```
Modified_data/vehicle_mass_props.py
 12 vehicle.mass_init.num_points = 1
 13 vehicle.mass_init.points = trick.sim_services.alloc_type( 1 , "jeod::MassPointInit" )
 14 vehicle.mass_init.points[0].name = "exercise 18 vehicle point"
```

```
Run_test/input.py
 33 dynamics.dyn_manager.add_body_action( vehicle.mass_init )
```

   (a) Position:

```
Modified_data/vehicle_mass_props.py
 15 vehicle.mass_init.points[0].position = trick.attach_units( "M",[-1.0, 0.0, 0.0])
```

   (b) There are three solutions provided for the orientation:

      i. Using an Eigen rotation about the z-axis (easiest)

```
Modified_data/vehicle_mass_props.py
 18 vehicle.mass_init.points[0].pt_orientation.data_source = trick.Orientation.InputEigenRotation
 19 vehicle.mass_init.points[0].pt_orientation.eigen_angle = trick.attach_units( "d",180)
 20 vehicle.mass_init.points[0].pt_orientation.eigen_axis = [0,0,1]
```

      ii. Using a transformation matrix

```
Modified_data/vehicle_mass_props.py
 22 vehicle.mass_init.points[0].pt_orientation.data_source = trick.Orientation.InputMatrix
 23 vehicle.mass_init.points[0].pt_orientation.trans[0]  = [ -1.0,  0.0, 0.0]
 24 vehicle.mass_init.points[0].pt_orientation.trans[1]  = [  0.0, -1.0, 0.0]
 25 vehicle.mass_init.points[0].pt_orientation.trans[2]  = [  0.0,  0.0, 1.0]
```

      iii. Using one entry of an Euler sequenced rotation.

```
Modified_data/vehicle_mass_props.py
 27 vehicle.mass_init.points[0].pt_orientation.data_source = trick.Orientation.InputEulerRotation
 28 vehicle.mass_init.points[0].pt_orientation.euler_sequence = trick.Orientation.YawPitchRoll
 29 vehicle.mass_init.points[0].pt_orientation.euler_angles = trick.attach_units( "d",[180.0, 0.0,
0.0])
```

3. See Modified_data/vehicle2_mass_props.py. Orientation only specified using the Eigen rotation method, but is otherwise the same.
4. Redundantly specified, zero rotation, axis is arbitrary.

```
Modified_data/vehicle_mass_props.py
  7 vehicle.mass_init.properties.pt_orientation.data_source = trick.Orientation.InputEigenRotation
  8 vehicle.mass_init.properties.pt_orientation.eigen_angle = 0.0
  9 vehicle.mass_init.properties.pt_orientation.eigen_axis  = [0.0, 1.0, 0.0]
```

5. Similar structure, this time with a non-zero angle.

```
Modified_data/vehicle2_mass_props.py
   8 vehicle2.mass_init.properties.pt_orientation.data_source = trick.Orientation.InputEigenRotation
   9 vehicle2.mass_init.properties.pt_orientation.eigen_angle =
trick.sim_services.attach_units( "d", 90.0 )
 10 vehicle2.mass_init.properties.pt_orientation.eigen_axis  = [0.0, 0.0, -1.0]
```

6. Still in the mass-properties file:

```
Modified_data/vehicle2_mass_props.py
  3 vehicle2.mass_init.properties.position = [0,0,0]
```

7. This requires that the *MassPoint* on vehicle 2 call the compute_relative_state method to compute its state. The

difficulty here arises because the *MassPoint* is not actually created until the *MassBodyInit* body-action goes to work, somewhere in the initialization process, after the input file. Consequently, the *MassPoint* does not exist when the input file is processed, so the method cannot be called from the the input-file directly. Furthermore, if the *MassPoint* instance does not exist until initialization, it certainly does not exist at simulation constructore, so the method cannot be called from the S_define either.

HINT:
The solution lies in the trick-scheduled events, or the "read" command.

SOLUTION:
The relative state is going to be written to a *MassPointState* instance, which does not exist in the simulation, so we had better add it. I added a tertiary simulation-object, called "*no_particular_name*" to provide this state (see S_define; ignore the instantiation of frame_state, that will be explored later). I wanted to put this in a simulation-object to make it loggable. I could have added it to the vehicle simulation object, but wanted to demonstrate its independence.
The *MassPoint* created by *MassBodyInit* can be identified by name, conveniently declared with the definition of the *MassPointInit*:

Modified_data/vehicle2_mass_props.py
```
15 vehicle2.mass_init.points[0].name = "vehicle2_point"
```

Now, in the input file, I can add a "read" statement to process this calculation once all of the initialization is complete. Here, I chose to make this calculation at the start of the simulation (0 seconds), but because the state is constant, that could have been at any time (before sim-shutdown).

RUN_test/input.py
```
43 trick.add_read(0, """
```

The triple quote here are a Python syntax allowing for the addition of multiple lines of code within one block.

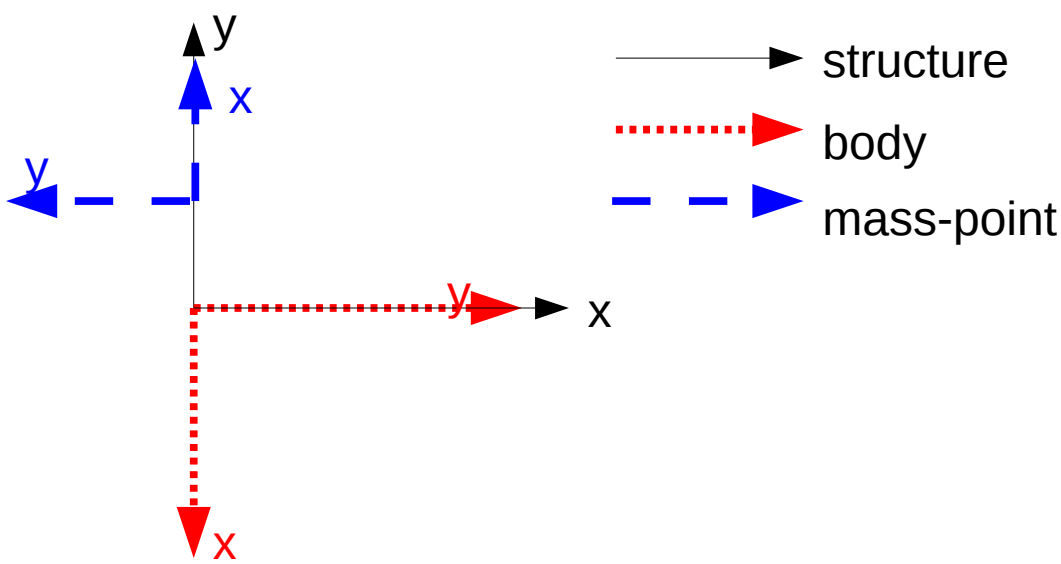
The function call can now be made inline. Note that it is not possible to call the *find_mass_point* and stash the pointer to the *MassPoint* somewhere because the SWIG interface cannot distinguish between pointers and references. Instead, we find the *MassPoint* based on the name, and in the same command, call its *compute_relative_state* method, passing in the vehicle's body-frame origin and the newly created *MassPointState* to store the result.

Notice the distinction between "vehicle2.dyn_body.mass.core_properties", and "vehicle2.dyn_body.core_body". The former defines a *MassPoint*, the latter a *BodyRefFrame*, which we will learn more about in Exercise 20. This particular method – called from a *MassPoint* – is for the generation of relative states between *MassPoint* instances.

For reference (beyond the scope of this exercise, and for interest only), I added a nearly-identical function call with another read statement. This one is called from a *RefFrame*, and computes the relative state of one *RefFrame* relative to another. This will all be explained in the next section, and explored in Exercise 20.

RUN_test/input.py
```
44 vehicle2.dyn_body.mass.find_mass_point("vehicle2_point").compute_relative_state (
45                                  vehicle2.dyn_body.mass.core_properties,
46                                  no_particular_name.point_state)
```

The results can now be logged, or pulled out of a run-time data handler, such as Trick-TV or a debugging routine. The diagram below illustrates the alignment of axes for vehicle2.



Consequently, the position of the *MassPoint* relative to the body-frame is at (-1.0, 0.0, 0.0), and the orientation is a 180-degree rotation about the z-axis:

$$T = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note – an application of this type of process is more commonly found in exterior models that use the JEOD base to provide frame-data. In this case, it would be appropriate to declare a *MassPoint* pointer in the exterior model, assign it to the mass point pointer produced by the *find_mass_point* function call, and then have that *MassPoint** available to call *compute_relative_state* as needed.

Pseudo-code example:

```
class ExteriorMode {
...
jeod::MassPoint * exterior_mass_point;
jeod::MassPointState point_state;
... }

ExteriorModel::initialize(...)
...
exterior_mass_point = vehicle2.dyn_body.mass.find_mass_point("vehicle2_point");
... }

ExteriorModel::run_time_process(...)
...
exterior_mass_point->compute_relative_state( ..., point_state );
.... }
```

Exercise 19 Interpreting the Mass Tree - Depleting a Tank Mass

Before getting into the solution for this simulation, it is worth spending a little time looking at the construction of the simulation itself. There are some interesting features included here.

S_define

The S_define starts as we are familiar – the two system simulation-modules, the default priority settings, time, and the non-dynamic-capable dynamics_init_only (since there are no dynamics in this simulation). Note that even though there are no dynamics in this simulation, the dynamics_init_only simulation-module includes the Dynamics Manager, which also manages attachments.

For the vehicle simulation-object, the default *vehicle_basic* simulation module seemed a poor fit for this simulation. It contains unnecessary content, and no tanks. I could add the tanks to a separate simulation-object (as the point_state was added in Exercise 18), but then the simulation-object would still contain unnecessary instantiations, so it got re-written.

First, notice that the simulation has no dynamics, so the vehicles need no state, so the *trans_init* and *rot_init* body actions are not needed. Also, there is no gravity, so the *grav_controls* are removed. A new *MassBody* was added, called *tank*, and the *mass_init* left intact.

With no dynamics in the simulation, the vehicle body (formerly the *DynBody* called *dyn_body*) can be replaced with a *MassBody* called, simply, *body.* This allows us to strip out the initialization of the *DynBody,* and therefore the necessity for having the Dynamics Manager passed into the vehicle simulation-object. That simplifies the constructor, and deletes the DynamicsManager reference.

Note – *DynBody* objects MUST be initialized with the Dynamics Manager, even if they have no dynamics. Hence, leaving this as a DynBody and just stripping out the initialization call and associated pass-in was not an option.

Notice that there is only one instance of *mass_init*; it can only initialize one *MassBody,* but there are two of those. The other necessary *MassBodyInit* will be created at the input-file level. In fact, the first one could also be created at the input-file level, but because one was already there, I left it. I could also have made this a 2-array of *MassBodyInit* objects, or 2 separate instances of *MassBodyInit* and had both available at the S_define level, but felt that illustrating the addition at the input-file level would be more constructive.

There is a final include provided to add the *BodyAttachAligned* option at the input-file level, we will need that, although it will not be covered until the next chapter.

Finally, because there is no dynamics, and nothing to integrate, the IntegLoop line has been deleted.

input file

Without the EmptySpace mode declaration, the mode would default to Ephemeris mode. That is no big deal, but I put the simulation on EmptySpace mode as a better descriptor of the scenario.

Even though no integration is needed, it is still necessary to specify an integrator.

The vehicles are only MassBody instances, they have no integration frame, nor dynamics (these lines are left in as comments for VehicleA to illustrate the difference).

The two vehicles are identical, with identical mass properties, so here I am illustrating the use of the python method for duplicate processes. Rather than create a mass-properties file for each vehicle, I create one, and use it for both by passing in the specific vehicle as an argument.

Mass-properties file (Modified_data/vehicle_mass_props.py)

I chose to overlay the four *MassBody* elements on top of one another to make the math easier. Consequently, all of the mass-points are at the origins of the respective bodies; attaching mass-point to mass-point puts all origins at the same location. Although the position specification is therefore not necessary (defaults to the zero-vector), it does serve to confirm the value as being zero. Specification of the inertia-tensor is necessary; it defaults to a zero-matrix, which is non-physical and could have serious consequences.

The vehicles are made symmetric, and orientation is irrelevant. However, it is recommended (to avoid Error messages) to specify an orientation data-source, even though it is not populated with data. With undefined orientations, and positions at the origin, the specification of how to interpret these data is unnecessary, so the *pt_frame_spec* assignment is gone.

The vehicle bodies both get two attach points – one for the other vehicle, and one for the tank (note that they are both at the same location, which isn't physical, but is allowable. It would also be possible to have the other vehicle and tank attach to a single point, but having the two points is a more realistic illustration of setting up such a scenario.

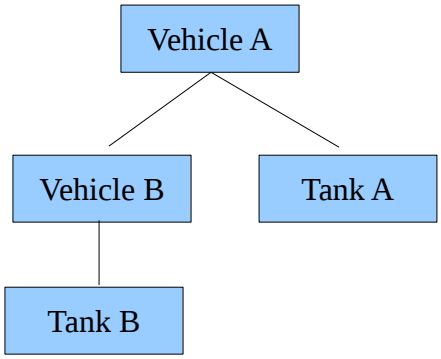
Because the two vehicles have identically-named mass-points, I can make the tank-to-vehicle attachment inside this method so that it will be processed for both vehicles with just one code block.

Attachment definition file (Modified_data/attach.py)

The vehicle-vehicle attachment should only be processed once (vehicleB attaches to vehicleA, not vice-versa) so that is processed outside the method defined in vehicle_mass_props.py (which gets called twice).

Solution

1.



The masses are as follows:

	Core mass / kg	Composite mass / kg
Vehicle A	10	40
Vehicle B	10	20
Tank A	10	10
Tank B	10	10

2. The print-tree command is executed through a “read” command for the same reasons that “read” was used in Exercise 18 – the input file has processed all of its regular commands before initialization jobs, and the body-action to combine the vehicles is performed as an initialization job. The file *initialization.out* confirms the data provided in the table above.
   Note also that the inertia tensors accumulate as masses are added.

3. Again, a read statement, with 4 commands:
    1. Change the mass
    2. Set the update flag on the changed mass (propagates up the tree).
    3. Update the mass properties of the root of the tree (propagates down the tree)
    4. Print the new tree.
4. The file *fifty_percent.out* prints the new mass-properties for the tree. These data are also logged.
   The only core-mass to change is that of *tankB.* The composite-mass of *vehicleA*, *vehicleB*, and *tankB* (but not *tankA)* also change, because the core-mass of *tankB* is included in those composite-masses.
   This process is repeated as more mass is removed from *TankB (not required).*

   NOTE – while the mass changed, the inertia tensor DID NOT. All we did was to change the mass of tank B, we did not change its inertia tensor. There is no automatic scaling of the inertia tensor with mass. In a real scenario, a separate scheduled model (rather than simple read statements) would handle the mass depletion, and that would be responsible for modifying the inertia tensor as a function of the remaining mass.

Exercise 20 Exercise with Mass Points and Body Ref Frames

1. Task. Note – I also changed the name of the simulation-object that contains the states fom *no_particular_object* to *states.*

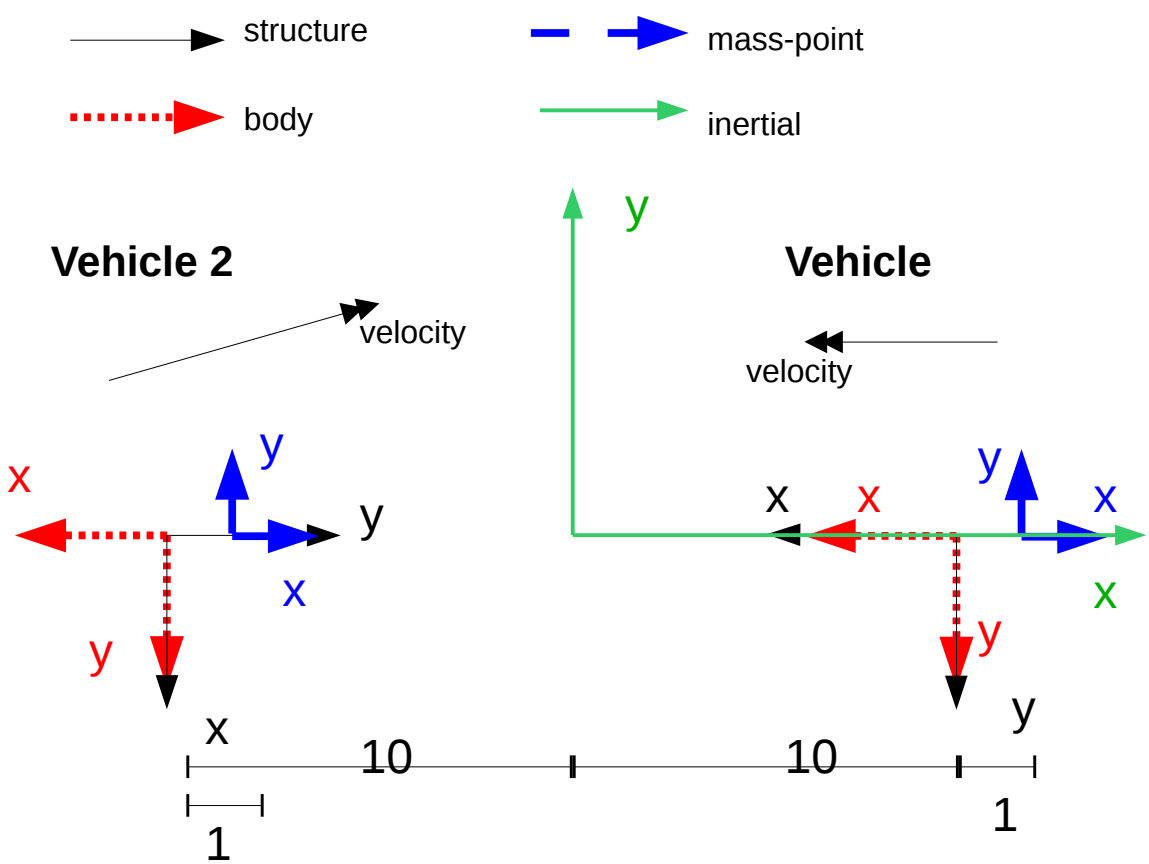
2. Edit *Modified_data/vehicle_state.py* and *Modified_data/vehicle2_state.py.* Change the *rot_init.ang_velocity* (line 14 in both files) values to zero.

3. Because the body-frames are a part of the *DynBody* class, they are created at construction, and this calculation can be put into the S_define as an initialization class job (since the question calls for the initial state only), or a run-time job (if the state is to be updated), However, following the method of Exercise 18, it is included again in the input file.

```
RUN_test/input.py
 47 vehicle2.dyn_body.core_body.compute_relative_state(
 48                                    vehicle.dyn_body.core_body,
 49                                    states.body_body_frame_state)
```

   Note that if the relative state were to be continually updated, it would be easier to call this from the S_define as a regularly scheduled job.
   The simulation-object *states* is given a *RefFrameState* instance called *body_body_frame_state* to provide the record of the state.

4. An illustration of the input data provides the following map to the initial configuration of the vehicles, their *MassPoint* instances, and their respective reference frames:



(a) From the diagram, it is apparent that relative to the body frame of *vehicle*, the frame associated

with the mass-point of *vehicle2* has state

position = (19.0, 0.0, 0.0) m

velocity = (-4.0, -1.0, 0.0) m/s

(b) Relative to the mass-point of *vehicle*, the frame associated with the mass-point of *vehicle2* has state:

position = (-20.0, 0.0, 0.0) m

velocity = (4.0, 1.0, 0.0) m/s

5. The same method concept is used to first find the vehicle-point, using find_vehicle_point, and then call its *compute_relative_state* method. Again, this is performed in the input file for convenience. Two additional frame states, *body_point_frame_state*, and *point_point_frame_state* are added to the *states* simulation_object.

RUN_test/input.py

```
vehicle2.dyn_body.find_vehicle_point("vehicle2_point").compute_relative_state(
                                        vehicle.dyn_body.core_body,
                                        states.body_point_frame_state)
vehicle2.dyn_body.find_vehicle_point("vehicle2_point").compute_relative_state(
                                        vehicle.dyn_body.find_vehicle_point("vehicle_point"),
                                        states.point_point_frame_state)
```

6. The variables are added to the Log_data file and compared to the predictions from step (4). They agree.

20.1 Sensors, Body Reference Frames, and Mass Points

1. Edited *vehicle_state.py* to reverse the change from the first part of this problem.
2. Task. Note that this is unusual; typically there will be only 1 Modified-data directory, with subdirectories to set the data for specific scenarios.
3.

vehicle2_mass_props.py

```
vehicle2.mass_init.points[0].name = “reflector”
```

4.

vehicle_mass_props.py

```
vehicle.mass_init.num_points = 3
vehicle.mass_init.points = trick.sim_services.alloc_type( 3 , "jeod::MassPointInit" )
vehicle.mass_init.points[0].name = "lidar-case"

vehicle.mass_init.points[1].name = "sun-sensor-case"

vehicle.mass_init.points[2].name = "imu-case"
```

5. In question 4.2 in the first part, we computed the state of vehicle2’s point relative to vehicle’s point. This is identical, only the names have changed:

```
vehicle2.dyn_body.find_vehicle_point("vehicle2_point").compute_relative_state(
                    vehicle.dyn_body.find_vehicle_point("vehicle_point"),
                                        states.point_point_frame_state)

   vehicle2.dyn_body.find_vehicle_point("reflector").compute_relative_state(
                                        vehicle.dyn_body.find_vehicle_point("lidar-case"),
                                        states.reflector_state)
    vehicle.dyn_body.integ_frame.compute_relative_state (
                                        vehicle.dyn_body.find_vehicle_point("sun-sensor-case"),
                                        states.sun_state)
    vehicle.dyn_body.find_vehicle_point("imu-case").compute_relative_state (
```

```
                                        vehicle.dyn_body.core_body,
                                        states.imu_body_state)
    vehicle.dyn_body.find_vehicle_point("imu-case").compute_relative_state (
                                        vehicle.dyn_body.integ_frame,
                                        states.imu_inrtl_state)
```

Note – in a real simulation environment, sensor models would be scheduled from the S_define; the sensor models would be executing these methods and post-processing the data. In particular, a sun-sensor model would not be interested in the distance (which is not easily measurable) but an angular displacement would be a more appropriate measure; this can be obtained from the true position and a little trigonometry.

Exercise 21 A New Two-vehicle Simulation

21.1 Mass Properties

1. Everything that is needed is available from the default simulation-modules. See the S_define for details.
2. See *Modified_data/vehicle_mass_props.py*. From the data, the body-frame origin is 3.0m from the structural origin, and on the +x axis. Therefore, the body-frame origin is at (3.0, 0.0, 0.0)m

```
3 vehicle.mass_init.properties.position = [3.0,0,0]
```

   The inertia tensor is easily obtainable: $I = 2.0\ kg\ m^2 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, provided on lines 5-7.

   The two frames are aligned, so the rotation is null (lines 8-10)

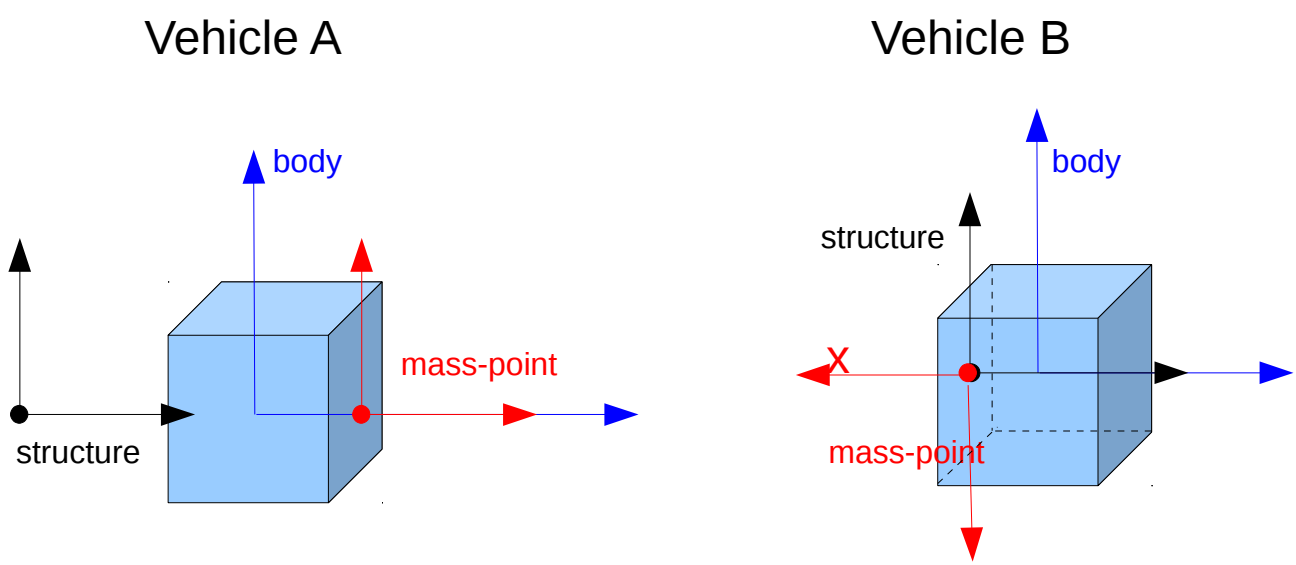
3. The mass-point is created and defined in lines 13-20
4. See *Modified_data/vehicleB_mass_props.py*. This time, the distance between frame origins is only 1.0m, with the body-frame still on the +x axis of the structure frame. The inertia scales with the mass, and the two frames are still aligned.

```
 3 vehicleB.mass_init.properties.position = [1.0, 0.0, 0.0]
```

5. Provide a structure-to-point orientation (actually, whether it is structure-to-point or point-to-structure is irrelevant in this special case) of 180-degrees yaw (z-axis).

```
 17 vehicleB.mass_init.points[0].pt_frame_spec    = trick.MassBasicPointInit.StructToPoint
...
 19 vehicleB.mass_init.points[0].pt_orientation.data_source = trick.Orientation.InputEigenRotation
 20 vehicleB.mass_init.points[0].pt_orientation.eigen_angle = trick.attach_units( "d",180.0)
 21 vehicleB.mass_init.points[0].pt_orientation.eigen_axis  = [0.0, 0.0, 1.0]
```

6.



21.2 Setting the Attach and Detach Processes

The attach and detach processes are defined in *Modified_data/attach_detach.py*. These steps have been explored before:

Create the attachment body-action instance and make it non-local:

```
mass_attach = trick.BodyAttachAligned()
mass_attach.thisown = 0
```

set the parent and subject and identify the mass-points

```
mass_attach.dyn_subject = vehicle.dyn_body
mass_attach.dyn_parent  = vehicleB.dyn_body
mass_attach.subject_point_name = "Vehicle A Mass Point 1"
mass_attach.parent_point_name  = "Vehicle B Mass Point 1"
```

Deactivate the attach process – we do not want this to occur immediately.

```
mass_attach.active = False
```

Add the body-action to the Dynamics Manager

```
dynamics.dyn_manager.add_body_action( mass_attach)
```

Create the detachment body-action instance and make it non-local:

```
mass_detach = trick.BodyDetach()
mass_detach.thisown = 0
```

set the subject being detached

```
mass_detach.dyn_subject = vehicle.dyn_body
```

Deactivate the detach process – we do not want this to occur immediately.

```
mass_detach.active = False
```

Add the body-action to the Dynamics Manager

```
dynamics.dyn_manager.add_body_action( mass_detach)
```

Notice that both body actions are deactivated. In the input file, read statements at 5.0 and 15.0 seconds activate the respective body-actions, and call the *perform_action* method.

```
trick.add_read(5.0 , """
mass_attach.active = True
dynamics.dyn_manager.perform_actions()
""")

trick.add_read(15.0 , """
mass_detach.active = True
dynamics.dyn_manager.perform_actions()
""")
```

21.3 Setting the Vehicle States and Testing

1. In the input file for part 3, the Modified-data files defining state are now included. From *Modified_data/vehicle_state.py*

```
(a)   1 vehicle.trans_init.dyn_subject              = vehicle.dyn_body
      2 vehicle.trans_init.reference_ref_frame_name = "Space.inertial"
      3 vehicle.trans_init.body_frame_id            = "composite_body"
      4 vehicle.trans_init.position                 = [0.0 , 0.0 , 0.0]
```

```
(b)   5 vehicle.trans_init.velocity                 = [0.0, 0.0, 0.0]
```

```
(c)   8 vehicle.rot_init.dyn_subject              =  vehicle.dyn_body
      9 vehicle.rot_init.reference_ref_frame_name = "Space.inertial"
```

```
10 vehicle.rot_init.body_frame_id            = "composite_body"
11 vehicle.rot_init.orientation.data_source  = trick.Orientation.InputEigenRotation
12 vehicle.rot_init.orientation.eigen_angle  = trick.sim_services.attach_units("d",-90.0)
13 vehicle.rot_init.orientation.eigen_axis      = [0, 0, 1]
```

```
(d) vehicle.rot_init.ang_velocity                = [0.0, 0.0, 0.0]
```

2. From *Modified_data/vehicleB_state.py:*

```
(a)   1 vehicleB.trans_init.dyn_subject              = vehicleB.dyn_body
      2 vehicleB.trans_init.reference_ref_frame_name = "Space.inertial"
      3 vehicleB.trans_init.body_frame_id            = "composite_body"
      4 vehicleB.trans_init.position                 = [-10.0, -2.0, 0.0 ]
```

```
(b)   5 vehicleB.trans_init.velocity                 = [ 2.0,  0.0, 0.0])
```

```
(c)   8 vehicleB.rot_init.dyn_subject              =  vehicleB.dyn_body
      9 vehicleB.rot_init.reference_ref_frame_name = "Space.inertial"
     10 vehicleB.rot_init.body_frame_id            = "composite_body"
     11 vehicleB.rot_init.orientation.data_source  = trick.Orientation.InputEigenRotation
     12 vehicleB.rot_init.orientation.eigen_angle  = trick.sim_services.attach_units("d",-90.0)
     13 vehicleB.rot_init.orientation.eigen_axis   = [0, 0, 1]
```

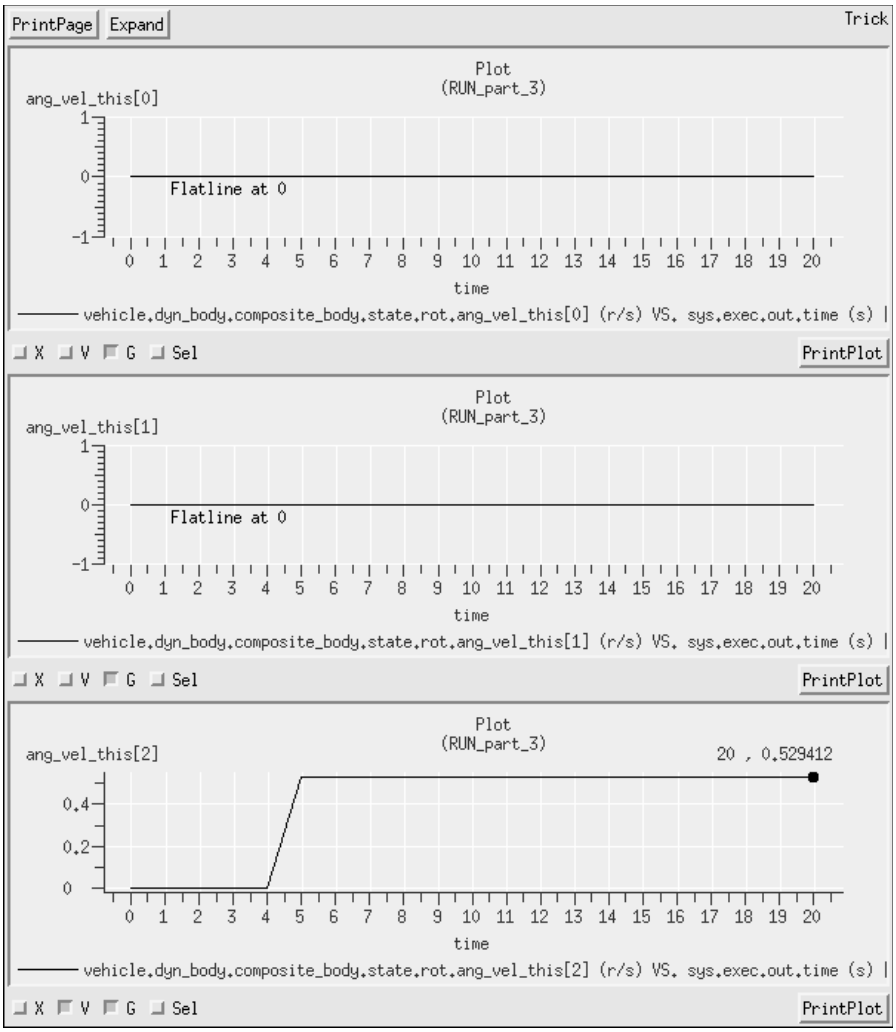
(d) Angular velocity is expressed in the frame that is being measured. So this describes the rate of the body frame relative to inertial, in the body-frame.

```
14 vehicleB.rot_init.ang_velocity = trick.sim_services.attach_units("d/s" , [20.0, 0.0, 0.0])
```

3. Need to consider the core and composite of vehicle B, and either one of vehicle A (they are identical for VehicleA). A little geometry indicates that at t=5.0 seconds, the two cubes are adjacent, with their mass-points in contact, and oriented appropriately for a seamless attachment.

4.

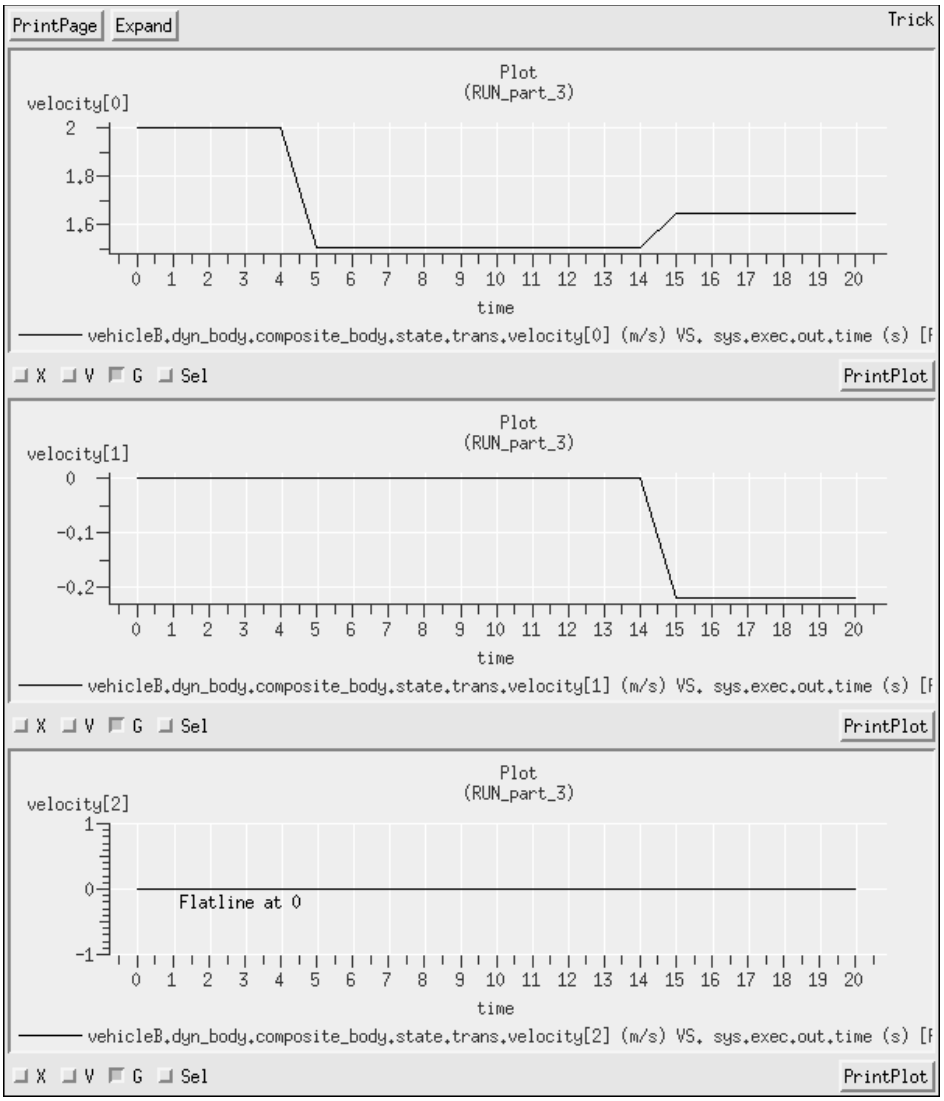
(a) Consider the angular rate. The new center of mass is 0.5m from the center of Vehicle B, 1.5m from the center of vehicleA. Considering the initial momentum of vehicleB, and the new inertia tensor of the combined vehicle, we obtain that both vehicles should start moving with an angular rate of 9/17 rad/s on the z-axis after attachment.



(b) Consider the angular rate on detach. There should be no changes in the angular rate of either vehicle on detach. This is observed on all three data sets.

(c) Consider the velocity on attach.

i. VehicleB is the parent body, so its composite-body value will represent the conservation of momentum. It should be moving at 18/12 = 1.5 m/s along the x-axis after attachment.



ii. The core velocity of vehicleB will have a value equal to the composite-body value + the rotation effect over a moment arm of 0.5m = . This will decrease, as the rotational effect reduces its contribution, and on the y-axis, the velocity will go positive as the combined vehicle rotates around counter-clockwise. Observed.

iii. The velocity of vehicleA will have a value equal to the composite-body value of vehicleB + the rotation effect over a moment arm of -1.5m = 24/34 m/s along the x-axis. This will increase (as rotational effect becomes less negative), and on the y-axis, the velocity will go negative as the combined vehicle rotates around counter-clockwise. Observed.

(d) Consider the velocity on detach. It is a function of the position of the vehicle, as both vehicles will continue to move with the 1.5 m/s along the x-axis, plus the tangential velocity of the rotational effect. After detach, the velocities will all remain constant. The velocity of vehicleA and the core-velocity of vehicleB should be continuous at detach; the composite velocity of vehicleB should be discontinuous since it measures the velocity of a discontinuous location on vehicleB. Observed.

(e) The position data should show continuous functions for the core-body values, with variations appropriate to the respective velocities. The position of the composite-body state of vehicleB should be discontinuous at both attach and detach as the center of mass shifts. Observed.

5. This produces some interesting results. With initial angular momentum on the inertial y-axis due to rotation (the body x-axis is parallel to the inertial y-axis), and on the z-axis due to the relative motion, the vehicle will rotate about an axis in the y-z plane, producing motion on all three axes.

Note – The attach/detach processes are handled quite differently between the input files for parts 2 and 3.

In part 2:

- Both processes are handled through BodyAction instances
- These are defined in the Modified-data file, *attach_detach.py*, which is executed from the input file
- They are activated in simple read statements in the input file.

In part 3:

- The attach process is still handled via a body-action, but this time the body-action is created, defined, and executed all within a single read statement.
- The detach process is handled directly, without use of a body-action. There is no detach body-action created.

Both methods work, or any combination of these methods. The difference is there purely as an illustration of how different architectures can yield the same results.

Exercise 22 Review of Exercise 19

VehicleA.body is unique in that both transformation matrices are identity. But this is the root of the tree; its structure-point's parent is inertial. Now, because the vehicle has no state, this orientation defaults to identity.
All bodies have identity for the core-properties transformation matrix. In the Mass Tree, the parent of core-properties is the vehicles's structure-point, and all vehicles were initialized with this orientation undefined (defaults to identity). Consequently, all *core_properties.T_parent_this* values are identity.

All bodies except *vehicleA.body* are attached to a parent using the *BodyAttachAligned* process. This process puts the two mass-points together with a relative yaw of 180 degrees. Because all mass-points are aligned with their respective bodies, all bodies are consequently yawed 180 degrees relative to their respective parent.

The parent of the structure-point of any body (other than the root vehicle, *vehicleA.body* in this case) is the parent's structure-point, so there must be a 180-degree yaw between each such pair. Consequently, all (except *vehicleA*) of the *structure_point.T_parent_this* matrices represent a 180-degree yaw.

Exercise 23 Initializing with Orbital Elements

1. S_define notes: Using the default S_modules, except for the introduction of the *DynBodyInitOrbit* class. For this, I included the header and instantiated it in the S_define. For rapid prototyping, I elected to keep the basic sim pattern from, for example, Exercise 8, and add a new sim-object to handle the state initialization. For a production simulation, I would have edited *vehicle_basic.sm* to replace the translational-state initializer with the *DynBodyInitOrbit* instance.

```
20 ##include "dynamics/body_action/include/dyn_body_init_orbit.hh"
21 class StateSimObject : public Trick::SimObject {
22    public:
23    jeod::DynBodyInitOrbit  orbit_init;
24    StateSimObject(){};
25    private:
26    StateSimObject (const StateSimObject&);
27    StateSimObject & operator = (const StateSimObject&);
28 };
29 StateSimObject state;
```

The data for the orbit initialization is provided in *Modified_data/vehicle_state_orbelm.py:*

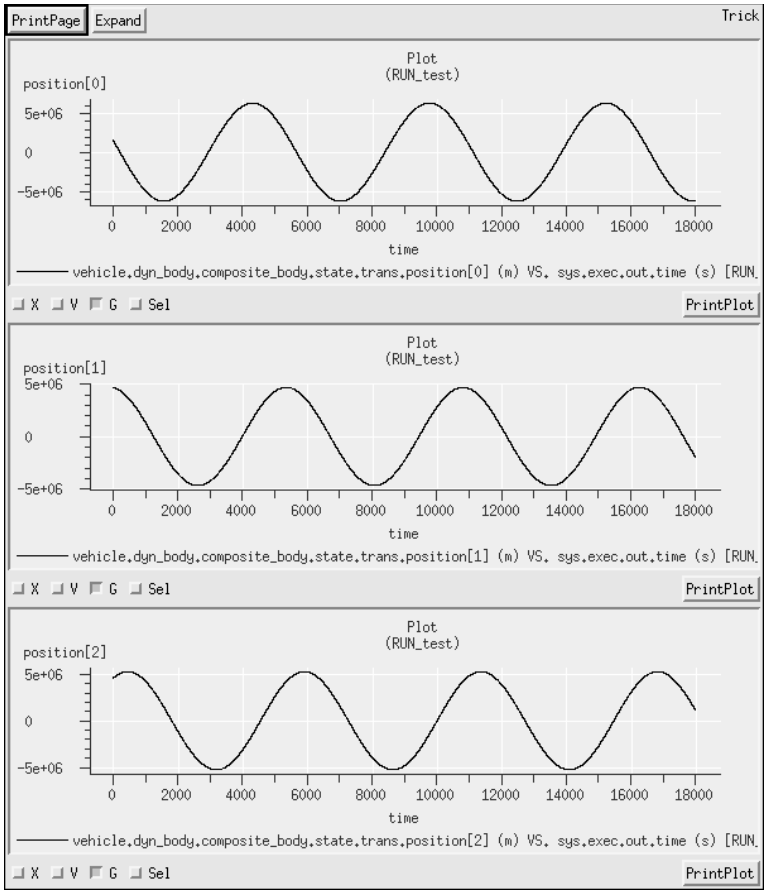
The subject of the body-action, the redundant assignment to reference_ref_frame_name, the reference-frame whose state is being initialized, the orbit-frame-name and the planet-name are all set. The choice of *set* is dictated by the available data in the exercise.

```
 1 state.orbit_init.dyn_subject              = vehicle.dyn_body
 2 state.orbit_init.reference_ref_frame_name = "Earth.inertial"
 3 state.orbit_init.body_frame_id            = "composite_body"
 4 state.orbit_init.set = trick.DynBodyInitOrbit.SmaEccIncAscnodeArgperTimeperi
 5 state.orbit_init.orbit_frame_name = "inertial"
 6 state.orbit_init.planet_name      = "Earth"
```

Note that the reference frame selected here is ECI *(Earth.inertial).* A valid alternative would be the planet-fixed frame fixed at Earth-center. In this simulation, we have a non-rotating Earth, so the two frames are identical, but in the more general case, the principle difference would be in the definition of the x-axis from which the longitude of ascending node is measured.

The file continues to defined the orbital elements as specified in the option selected on line 4.

```
 8 state.orbit_init.semi_major_axis  = trick.sim_services.attach_units("km" , 6700.0)
 9 state.orbit_init.inclination      = trick.sim_services.attach_units("d" , 52.0)
10 state.orbit_init.eccentricity     = 0.01
11 state.orbit_init.arg_periapsis    = trick.sim_services.attach_units("d" , 60.0)
12 state.orbit_init.ascending_node   = trick.sim_services.attach_units("d" , 25.0)
13 state.orbit_init.time_periapsis   = 0.0
14
15 dynamics.dyn_manager.add_body_action( state.orbit_init )
```



Exercise 24 Initializing from LVLH

1. Task
2. Simply instantiate a second instance, as we have done before. For the mass properties and gravity controls, I converted the respective files from SIM_23 into function-definitions, and called the new functions for both vehicle and vehicle2 so that both vehicles are initialized identically.
3. Notice that the reference body for the translational and rotational initializations are different. An instance of *DynBodyInitLvlhState* can only have one reference body. We could use two instance of the *DynBodyInitLvlhState*, but would have to down-select the states that each initializes (the default setting for *DynBodyInitLvlhState* is to initialize both the translational and rotational states). Alternatively, we could use the classes specifically intended for translational and rotational states only: *DynBodyInitLvlhTransState* and *DynBodyInitLvlhRotState*.

   The simplest solution to this problem would be to just add instances of these state initializers to the State simulation object. However, I also want to demonstrate the usage of these state-initializing body-actions when they are dynamically allocated, so the StateSimObject can now be removed. Note, however, that the header files must remain included.

```
S_define
##include "dynamics/body_action/include/dyn_body_init_orbit.hh"
##include "dynamics/body_action/include/dyn_body_init_lvlh_rot_state.hh"
##include "dynamics/body_action/include/dyn_body_init_lvlh_trans_state.hh"
```

This implementation utilizes the capability to create and manage the necessary *LvlhFrame* behind-the-scenes without instantiating one in the S_define. Notice that this is handled internally; it is not even necessary to include the header-file for the *LvlhFrame*.

In the Modified data file, make a new instance of DynBodyInitLvlhTransState, and set the values as specified.

```
Modified_data/vehicle2_state_lvlh.py
lvlh_trans = trick.DynBodyInitLvlhTransState()
lvlh_trans.thisown = 0

lvlh_trans.dyn_subject   = vehicle2.dyn_body
lvlh_trans.body_frame_id = "composite_body"
lvlh_trans.ref_body_name = "veh"
lvlh_trans.planet_name   = "Earth"
lvlh_trans.position  = [-200.0, 0.0, 0.0]
lvlh_trans.velocity  = [  10.0, 0.0, 0.0]
...
dynamics.dyn_manager.add_body_action( lvlh_trans )
```

This states that we are initializing the “composite_body” frame of “vehicle2.dyn_body” using an LVLH frame attached to the vehicle with name “veh”, using “Earth” to define vertical and horizontal.

4. Use the *DynBodyInitLvlhRotState:*

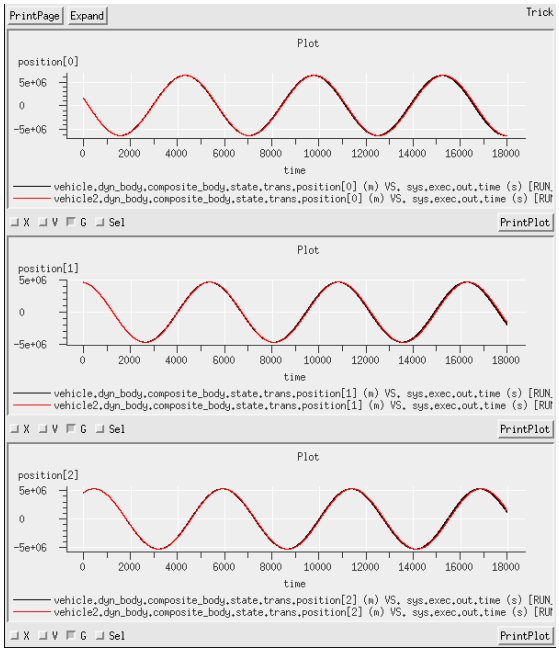
In Modified-data, initialize the composite-body frame of vehicle2.dyn_body using an LVLH frame attached to the same body (i.e. vehicle2.dyn_body), using Earth to define vertical and horizontal. Note that in this case, we do not need to specify the *ref_body_name* because it will default to itself.

```
Modified_data/vehicle2_state_lvlh.py
lvlh_rot = trick.DynBodyInitLvlhRotState()
lvlh_rot.thisown = 0

lvlh_rot.dyn_subject    =  vehicle2.dyn_body
lvlh_rot.body_frame_id  = "composite_body"
lvlh_rot.planet_name    = "Earth"
lvlh_rot.orientation.data_source    = trick.Orientation.InputEulerRotation
lvlh_rot.orientation.euler_sequence = trick.sim_services.Yaw_Pitch_Roll
lvlh_rot.orientation.euler_angles   = trick.sim_services.attach_units("d", [0.0, 90.0, 0.0])
lvlh_rot.ang_velocity               = [0.0, 0.0, 0.0]
...
dynamics.dyn_manager.add_body_action( lvlh_rot )
```

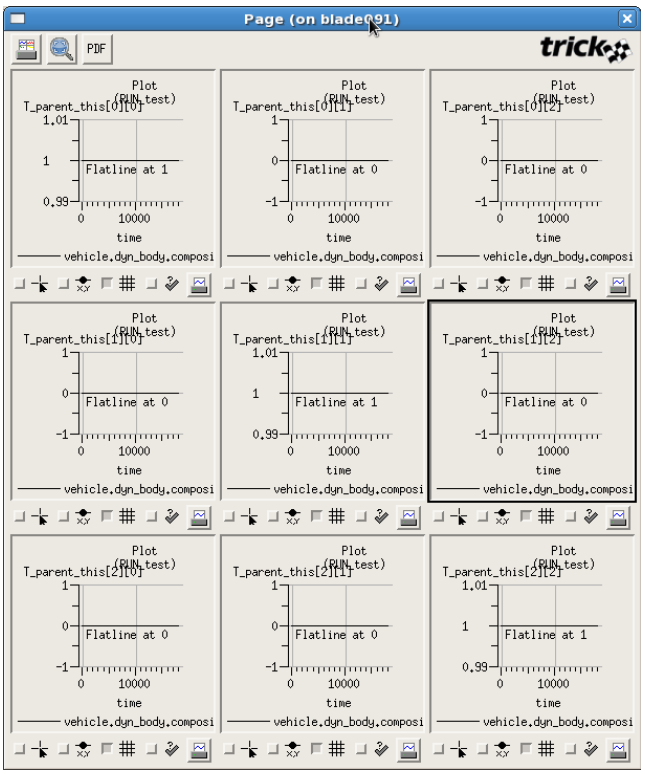
5. The *vehicle.sm* simulation module already includes a rotational-state initialization body-action; it is only necessary to populate this and register it.

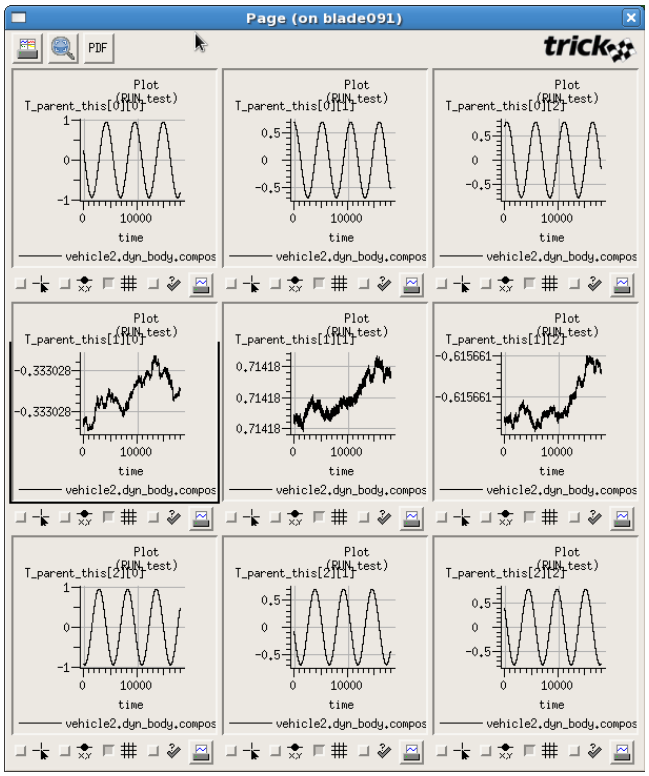
```
Modified_data/vehicle_state_orbelm.py
vehicle.rot_init.dyn_subject              =  vehicle.dyn_body
vehicle.rot_init.reference_ref_frame_name = "Earth.inertial"
vehicle.rot_init.body_frame_id            = "composite_body"
vehicle.rot_init.orientation.data_source  = trick.Orientation.InputEulerRotation
vehicle.rot_init.orientation.eigen_angle  = 0.0
vehicle.rot_init.orientation.eigen_axis   = [0, 0, 1]
vehicle.rot_init.ang_velocity             = [0.0, 0.0, 0.0]
...
dynamics.dyn_manager.add_body_action( vehicle.rot_init )
```

6. Position stays close, gradually diverging as would be expected by two vehicles in slightly different orbits. By the end of the simulation, the two vehicles are about 500km apart.



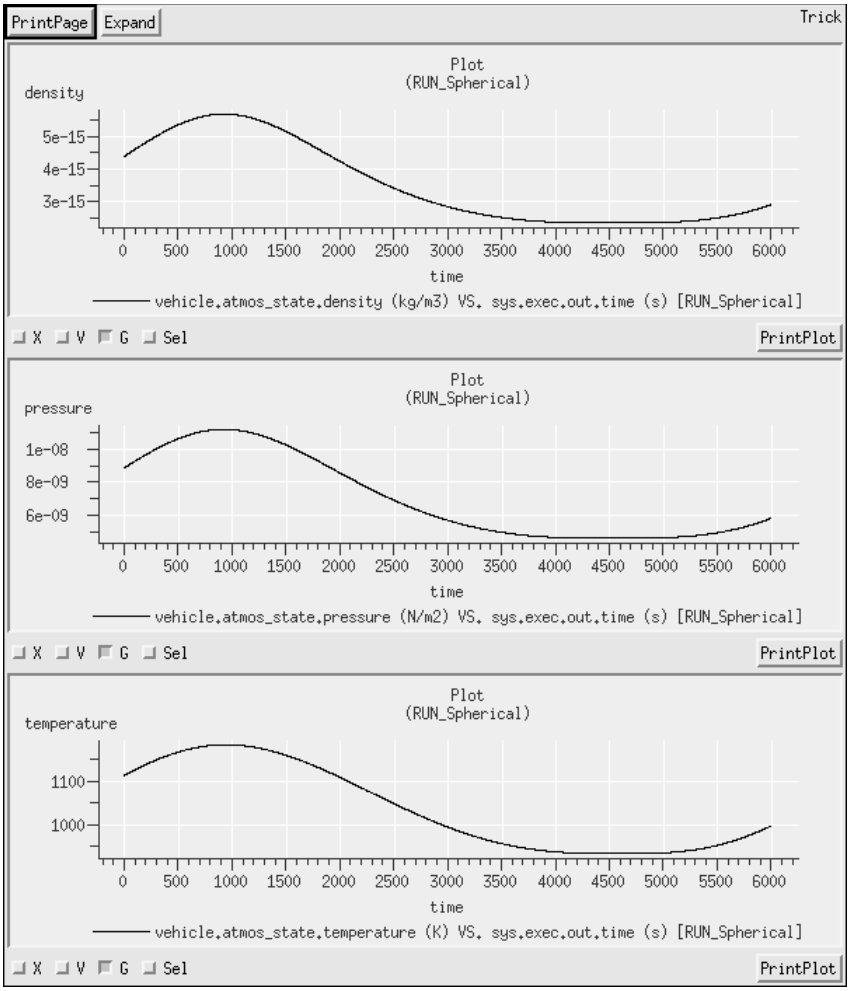
The attitude is more interesting. The original vehicle was initialized with no angular velocity relative to the inertial reference-frame, so relative to the inertial reference-frame, it remains static. Conversely, the new vehicle was initialized with no angular velocity relative to an LVLH frame, which is itself rotating at a rate of 1 revolution per orbital-period. Consequently, the second vehicle will rotate at one revolution per orbital period relative to the inertial reference-frame. The transformation matrices from inertial to body are shown:





Exercise 25 Add an Atmosphere

1. Task. Note, I also changed the name of the state definition file from *vehicle_state_ex8.py* to *vehicle_state.py*.
2. Task
3. Replaced the long Earth simulation-object with earth_GGM02C_MET_RNP.sm
4. Replaced vehicle_basic.sm with vehicle_atmosphere.sm
5. For the vehicle states I copied in from SIM_12, the vehicle is very close to a circular orbit. The temperature, density and pressure all vary significantly. This is unlikely to be due to changes in altitude.



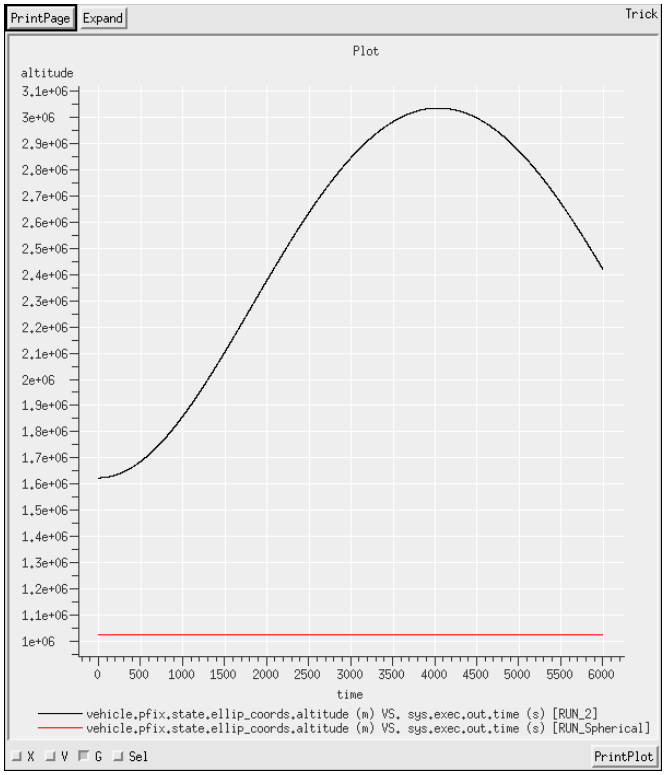
However, the initial longitude is at 2.8 radians, and the initial time is midnight UTC. That makes the local time at the vehicle about 10:40 am. As the vehicle advances in longitude, it is moving closer to noon, the temperature, density, temperature values should be increasing.

6. I made a second run, RUN_2 based entirely on RUN_Spherical but with an increase to the initial height of the orbit.

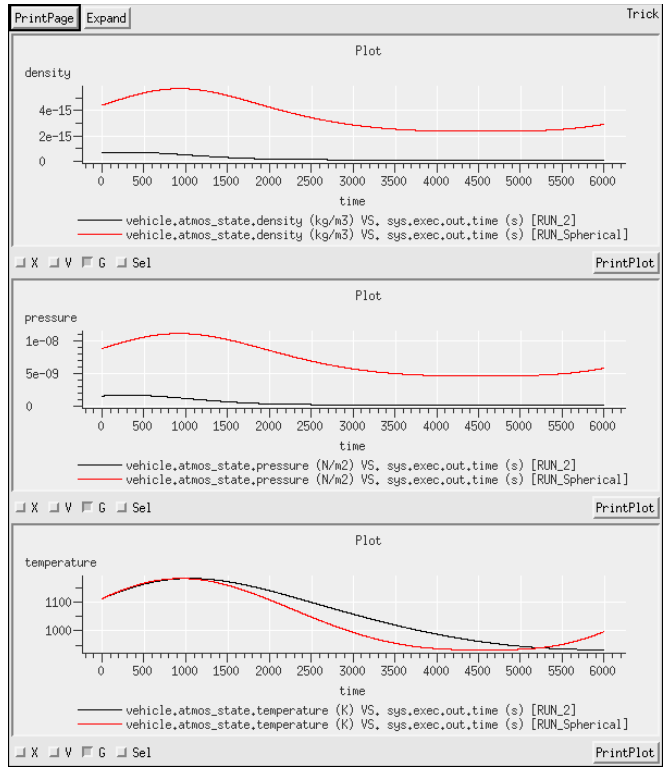
RUN_2/input.py

```
execfile("RUN_Spherical/input.py")
vehicle.trans_init.position  = [8000000.0, 0.0, 0.0 ]
```

7. That makes the orbit more eccentric, and wholly above the previous orbit. The vehicle starts close to periapsis. With the increased altitude, the pressure and density should fall. The temperature variation is not obvious. However, the vehicle should continue to move past noon and around to the dark side; the same variations should be seen only more slowly as the orbital period has increased. Data support predictions.



Altitude variation with time



ρ, P, T variation with time.

Exercise 26 Setting up a Surface Model

1. Task.
2. Two choices here:

   (a) Expand vehicle_atmosphere.sm and insert a surface model,

   (b) Create a new simulation object that defines the surface model.

   For rapid prototyping, option (b) is probably the faster and cleaner solution, with less potential for typographical errors. For integration into an existing simulation, option (a) would be the better solution for ease of long-term maintenance. The S-define in the solutions uses option(b). The instructions for option (a) are as follows:

   i. Include the headers for the surface model and for the desired facets (in this case, flat plate).

   ii. Expand *vehicle_atmosphere.sm*, and add an instance of a *SurfaceModel* to the expanded *vehicle_atmosphere.sm*

   The instructions for option (b) are as follows:

i. include the headers for the surface model and for the desired facets (in this case, flat plate).

ii. Create a new simulation-module that contains a single element, an instance of a *SurfaceModel*.

```
S_define
 20 ##include "utils/surface_model/include/flat_plate.hh"
 21 ##include "utils/surface_model/include/surface_model.hh"
...
 28    jeod::SurfaceModel          surface;
```

3. Made a new modified-data file, *surface.py* to define the surface. For convenience, all facets were placed on the primary axes, positioned and sized according to the exercise specifications.

4. Compared the same plots as illustrated in the solutions for Exercise 25. Difference-plots showed no change.

Exercise 27 Add Aerodynamic Drag

1. Task.

2. The interactions simulation-object is found at lines 20-58. Note – in this example, I moved my *SurfaceModel* from its single-element simulation-object and added it to the newly created *InteractionSimObject*. Whether the *SurfaceModel* belongs with the vehicle sim-object, or with the interactions sim-object, or whether the vehicle, surface and interactions should all be in one sim-object is a matter of personal preference. Operationally, there is no difference, but be careful that the instances are correctly passed and addressed (e.g. Line 42 passes the surface-model to the aero-surface-factory; if the surface resides in another sim-object, this sim-object would need a reference to it or to its simulation-object from which it can be accessed)

3.

```
 25 ##include "interactions/aerodynamics/include/aero_drag.hh"
...
 33   jeod::AerodynamicDrag aero_drag;
```

4. The AeroSurface is instantiated and created in the interactions simulation-object.

```
 26 ##include "interactions/aerodynamics/include/aero_surface.hh"
 27 ##include "interactions/aerodynamics/include/aero_surface_factory.hh"
 28 ##include "interactions/aerodynamics/include/flat_plate_aero_params.hh"
...
 34   jeod::AeroSurface aero_surface;
 35   jeod::AeroSurfaceFactory aero_surface_factory;
...
 41       P_BODY ("initialization") aero_surface_factory.create_surface(
 42                                                          &surface,
 43                                                          &aero_surface);
```

5. I'm using the default simulation modules throughout, so I pulled these straight from the course notes.

```
 45       (DYNAMICS, "scheduled")  aero_drag.aero_drag(
 46                          vehicle.dyn_body.composite_body.state.trans.velocity,
 47                         &vehicle.atmos_state,
 48                          vehicle.dyn_body.structure.state.rot.T_parent_this,
 48                          vehicle.dyn_body.mass.composite_properties.mass,
 50                          vehicle.dyn_body.mass.composite_properties.position);
```

6.

```
 60 vcollect vehicle.dyn_body.collect.collect_environ_forc jeod::CollectForce::create {
 61            interactions.aero_drag.aero_force
```

```
62 };
63 vcollect vehicle.dyn_body.collect.collect_environ_torq jeod::CollectTorque::create {
64            interactions.aero_drag.aero_torque
65 };
```

7. The aero-parameters for the surface are provided in the Modified-data surface file. Before creating instances on-the-fly, be sure to include the appropriate header file.

```
S_define
##include "interactions/aerodynamics/include/flat_plate_aero_params.hh"
```

(a)

```
Modified_data/surface.py
56 params = trick.FlatPlateAeroParams()
57 params.thisown = 0
58 params.name = "black"
59 params.coef_method = trick.AeroDragEnum.Diffuse
60 params.calculate_drag_coef = True
61 interactions.aero_surface_factory.add_facet_params(params)
```

(b)

```
63 params = trick.FlatPlateAeroParams()
64 params.thisown = 0
65 params.name = "white"
66 params.coef_method = trick.AeroDragEnum.Mixed
67 params.epsilon = 0.8
68 params.calculate_drag_coef = True
69 interactions.aero_surface_factory.add_facet_params(params)
```

Because these parameters include the requirement to calculate the drag coefficients (lines 60, 68), we need to set the free-stream temperature and gas constant values. These are done in the input file (which will soon be moved to *RUN_aero/input.py*)

```
RUN_aero/input.py
  6 interactions.aero_drag.param.gas_const = 287
  7 interactions.aero_drag.param.temp_free_stream = 1487
```

8. Task

9. Use one as the basis, call it from the other, and add the deltas. Because the aero-drag model defaults *"on"*, the grav-only run needs to turn it off. Because only the *RUN_aero* run uses aerodynamic parameters, those only need setting in *RUN_aero/input.py*. Because there are more lines to set for RUN_aero, I chose to make RUN_gravonly the basis. That does result in turning aero off and immediately back on again, butt here has to be a trade-off somewhere.

   Thus, in this solution, *RUN_gravonly* became the basis input file with aero turned off, and *RUN_aero* added the deltas to turn the aerodynamic drag back on. There are equally viable alternatives.

   (a)

```
RUN_gravonly/input.py
35 interactions.aero_drag.active = False
```

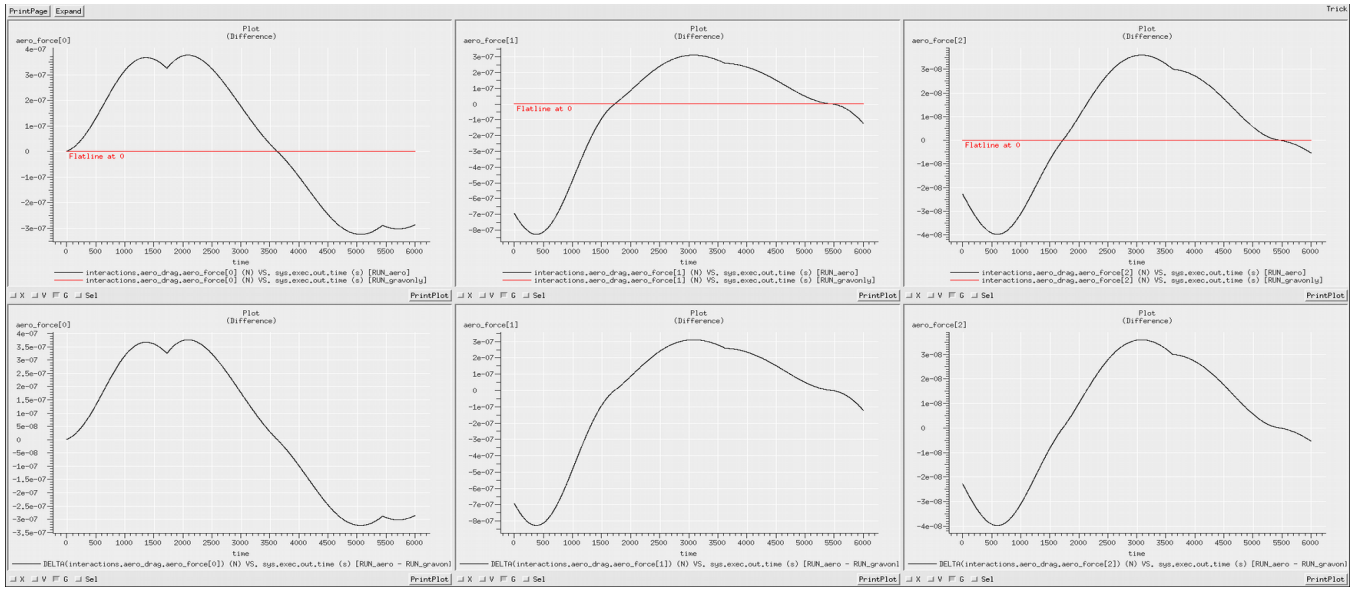
(b)

```
RUN_aero/input.py
  1 execfile("RUN_gravonly/input.py")
  2 interactions.aero_drag.active = True
  3 interactions.aero_drag.use_default_behavior = False
  4 interactions.aero_drag.set_aero_surface(interactions.aero_surface)
```

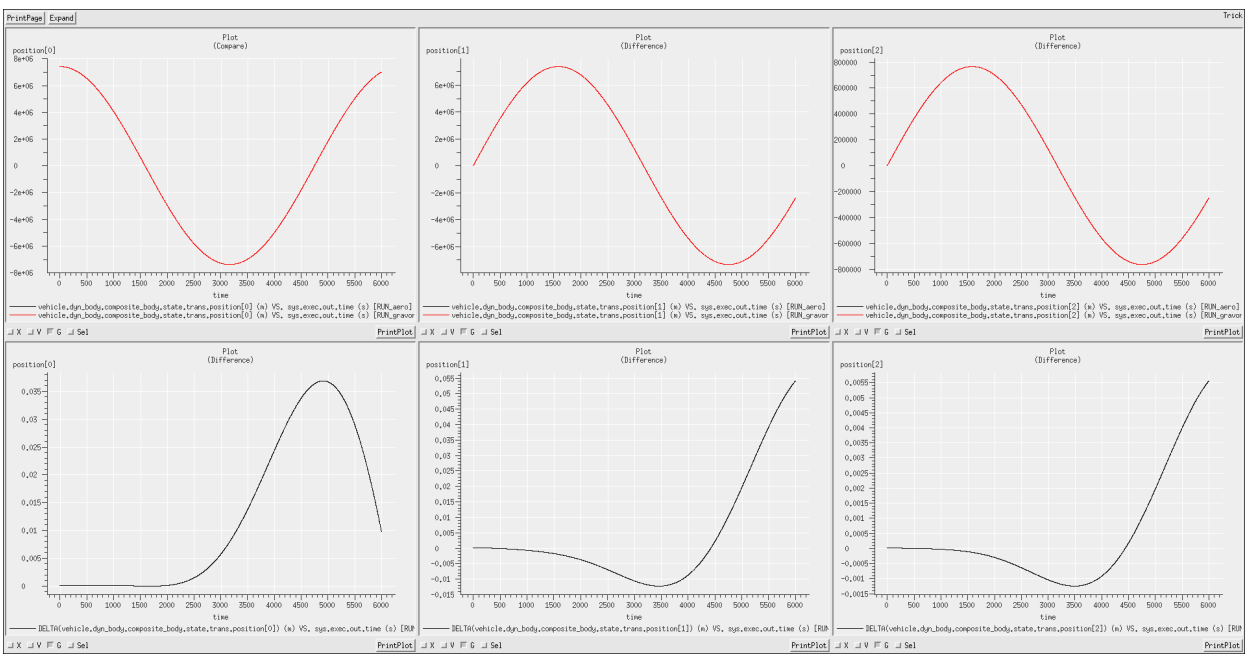
10. Task

11. Task

12. The applied forces. The forces in *RUN_gravonly* (red, upper panes) flatline at 0. The deltas (lower panes) are copies of the RUN_aero forces ((black, upper panes)



The vehicle positions. Over 1 orbit of a compact massive object, the effects are small, just centimeters on each axis.



Exercise 28 Add Radiation Pressure

1. Task
2. Multiple components to adding the Radiation Pressure Model, including an expansion to the constructor arguments. The Radiation Pressure model includes modeling of facet temperatures, so the *FlatPlate* gets upgraded to a *FlatPlateThermal*.

```
26 ##include "utils/surface_model/include/flat_plate_thermal.hh"
```

Then the multiple components get added:

```
33 ##include "interactions/radiation_pressure/include/radiation_pressure.hh"
34 ##include "interactions/radiation_pressure/include/radiation_surface.hh"
35 ##include "interactions/radiation_pressure/include/radiation_surface_factory.hh"
36 ##include "interactions/radiation_pressure/include/radiation_third_body.hh"
37 ##include "interactions/radiation_pressure/include/radiation_params.hh"
...
48   jeod::RadiationPressure       rad_pressure;
49   jeod::RadiationSurface        rad_surface;
50   jeod::RadiationSurfaceFactory rad_surface_factory;
```

The new function calls:

```
66       P_ENV  ("initialization") rad_surface_factory.create_surface(
67                                                        &surface,
68                                                        &rad_surface);
69
70       P_ENV  ("initialization") rad_pressure.initialize(
71                                  dyn_mngr,
72                                 &rad_surface,
73                                  vehicle.dyn_body.mass.composite_properties.position);
...
83       (DYNAMICS, "scheduled")  rad_pressure.update(
84                                vehicle.dyn_body.structure,
85                                vehicle.dyn_body.mass.composite_properties.position,
86                                dyn_time.scale_factor,
87                                dyn_time.seconds);
```

Add the external references to the simulation-object and populate them:

```
96   jeod::TimeDyn             & dyn_time;
97   jeod::DynManager          & dyn_mngr;

54   InteractionSimObject( VehicleAtmSimObject  & veh_in,
55                         jeod::TimeDyn              & dyn_time_in,
56                         jeod::DynManager           & dyn_mngr_in)
...
59        dyn_time(dyn_time_in),
60        dyn_mngr(dyn_mngr_in)
...
99 InteractionSimObject interactions(vehicle,
100                                  jeod_time.time_manager.dyn_time,
101                                  dynamics.dyn_manager);
```

3. See lines 105, 110

```
103 vcollect vehicle.dyn_body.collect.collect_environ_forc jeod::CollectForce::create {
104            interactions.aero_drag.aero_force,
105            interactions.rad_pressure.force
106 };
107
108 vcollect vehicle.dyn_body.collect.collect_environ_torq jeod::CollectTorque::create {
109            interactions.aero_drag.aero_torque,
110            interactions.rad_pressure.torque
111 };
```

4. The Radiation Data Recorder allows useful insight into the inner-workings of the dynamically allocated facets. It is optional.

```
39 ##include "interactions/radiation_pressure/verif/include/radiation_data_recorder.hh"
...
52   jeod::RadiationDataRecorder  rad_data_rec;
...
89       (DYNAMICS, "scheduled") rad_data_rec.record_data ( &rad_surface);
```

5. Adding the sun means going to Ephemerides mode (input file change), and changing from *environment_sans_de405.sm* to *environment.sm.* Furthermore, because Earth does not "parallel" Sun in the reference-frame tree, we actually need Earth-Moon Barycenter, and the easiest way to get that is to add Moon.

```
16 #include "environment.sm"

18 #include "sun_basic.sm"
19 #include "moon_basic.sm"
```

6. See Modified_data/surface.py. Trivial change.

7.

```
Modified_data/surface.py
79 params = trick.RadiationParams()
80 params.thisown = 0
81 params.name = "black"
82 params.albedo = 0.2
83 params.diffuse = 0.95
84 params.thermal.emissivity = 0.8
85 params.thermal.heat_capacity_per_area = 200.0
86 interactions.rad_surface_factory.add_facet_params(params)
87
88 params = trick.RadiationParams()
89 params.thisown = 0
90 params.name = "white"
91 params.albedo = 0.7
92 params.diffuse = 0.7
93 params.thermal.emissivity = 0.3
94 params.thermal.heat_capacity_per_area =  500.0
95 interactions.rad_surface_factory.add_facet_params(params)
```

8. See Modified_data/surface.py, lines 8, 18, 28, etc.

```
8 flat_plate.thermal.active = True
```

9. This command was already issued as part of establishing the Radiation Pressure Model (part 2, line 66 in S_define)

10. Because RUN_aero imports RUN_gravonly, need only change in RUN_gravonly.

```
RUN_gravonly/input.py
35 interactions.rad_pressure.active = False
```

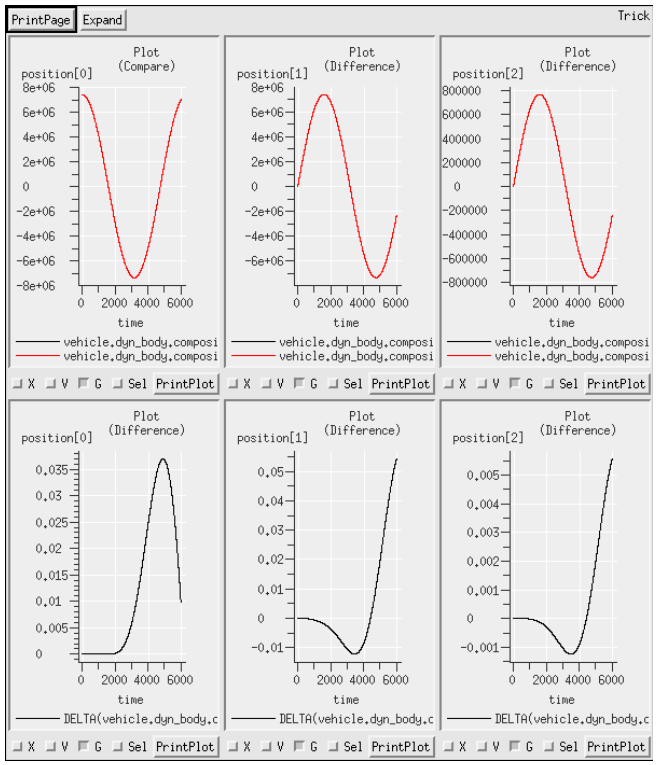
11. Plotted deltas on position, found no difference.

12. Made *RUN_rad* import *RUN_gravonly*, and turned on radiation pressure. Made *RUN_both* import *RUN_aero*, and likewise, turned on radiation pressure. See respective input files.

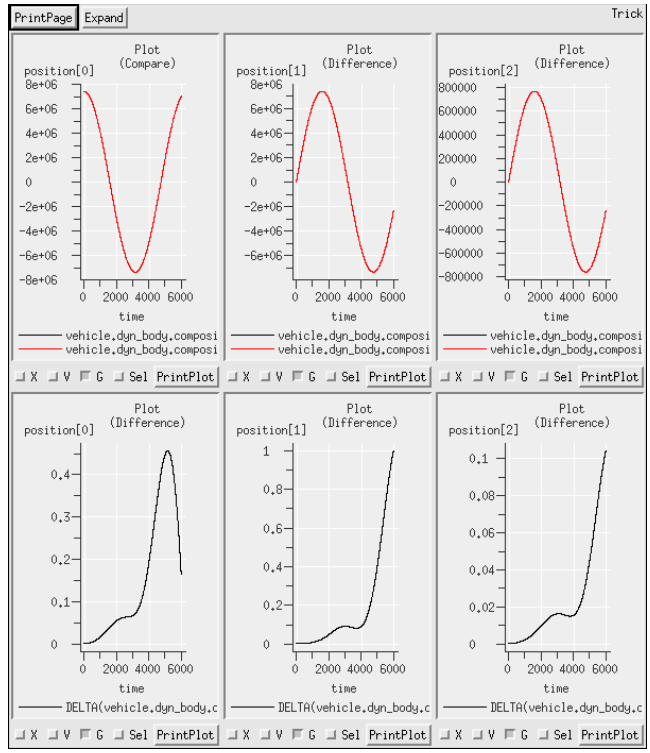
13. Task

14. Position results of 3 runs compared against the baseline RUN_gravonly:

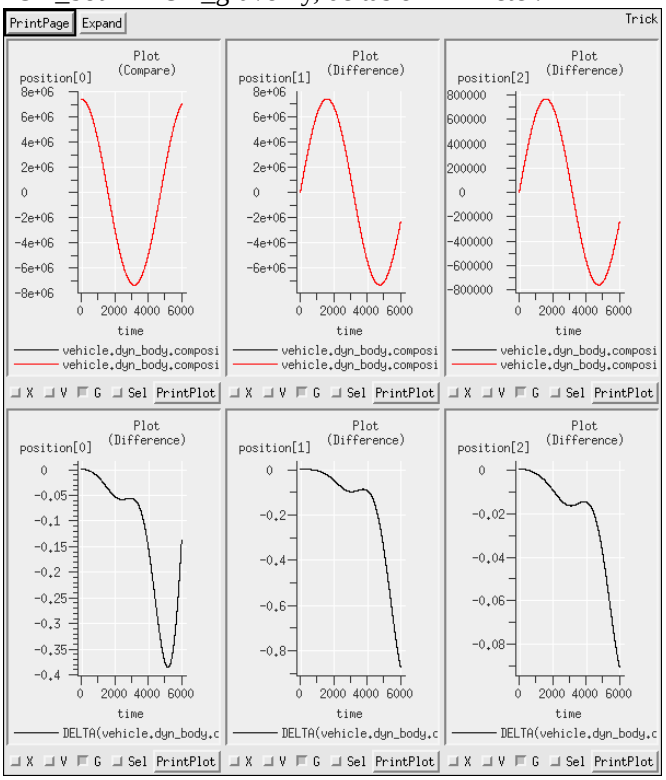
RUN_aero - RUN_gravonly , deltas of centimeters.



RUN_gravonly - RUN_rad, deltas of ~ 1meter.

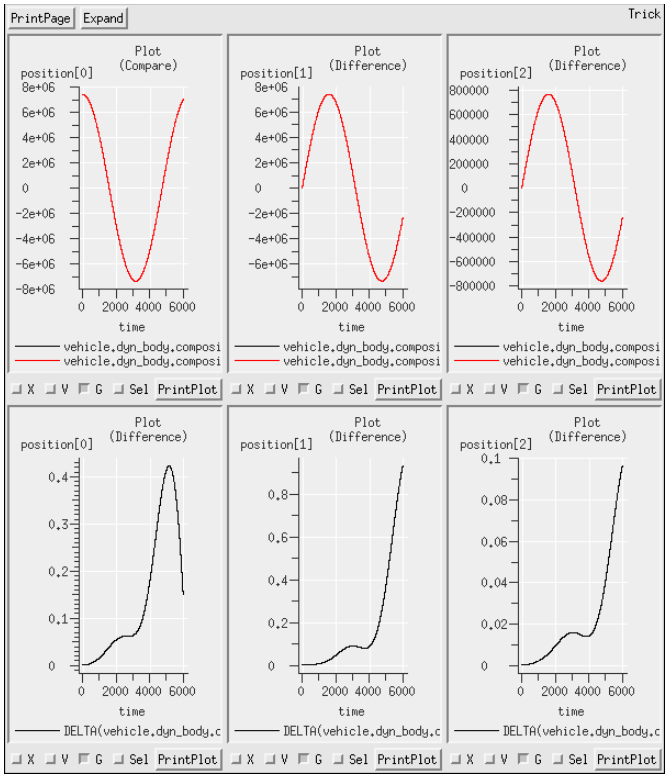


RUN_both - RUN_gravonly, deltas of ~ 1 meter.



RUN_aero – RUN_both.

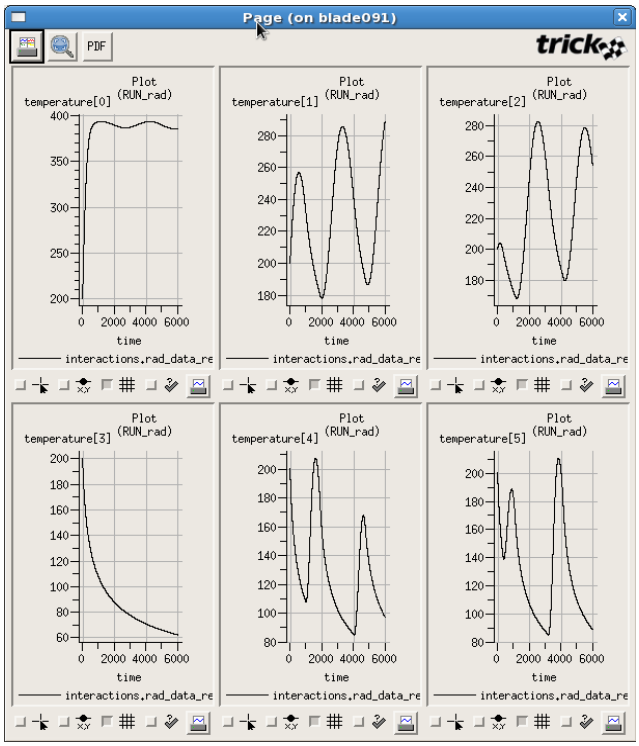
This also shows the effect of (-) radiation pressure



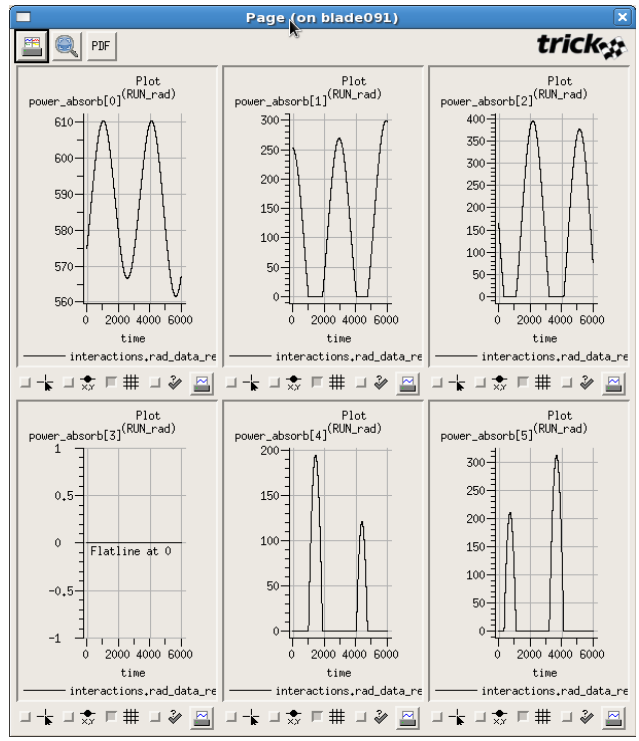
15. The facet temperatures are directly related to the amount of sunlight being absorbed. Notice that the opposing facets (in each column) are never illuminated at the same time. Adjusting the rotation rate will produce different profiles, the following profiles were obtained with an initial angular rate of:

```
vehicle.rot_init.ang_velocity        = [0.002, 0.0002, 0.0003]
```

Facet temperature



Facet power absorbed



Exercise 29 Utilizing the Contact Model

1. We have seen this configuration in several previous examples.

```
 7 #define DYNAMICS      1.0
 8
 9 #include "sim_objects/default_trick_sys.sm"
10 #include "jeod_sys.sm"
11 #include "default_priority_settings.sm"
12 #include "time_dyntime_only.sm"
13 #include "dynamics.sm"
14 #include "vehicle_basic.sm"
15 VehicleSimObject vehicleB(dynamics.dyn_manager);
```

2. I made *vehicle* stationary, and *vehicleB* move towards it on the y-axis. See Modified_data/vehicle_state.py and Modified_data/vehicleB_state.py.

3. Made the plate on *vehicle* point along the -y-axis, and the plate on *vehicleB* point along the +y-axis

Modified_data/surface.py

```
 1 fpc = trick.FlatPlateCircular()
 2 fpc.thisown = 0
 3 fpc.position  = trick.attach_units( "M",[ 0.0 , 0.0, 0.0 ])
 4 fpc.normal  = [ 0.0 , -1.0 , 0.0 ]
 5 fpc.radius  = trick.attach_units( "M",1.0)
 6 fpc.param_name = "steel"
 7 fpc.mass_body_name = "veh"
 8 fpc.name = "collision face"
 9
10 interactions.veh_surface.add_facet(fpc)
11
12
13 fpc = trick.FlatPlateCircular()
14 fpc.thisown = 0
15 fpc.position  = trick.attach_units( "M",[ 0.0 , 0.0, 0.0 ])
16 fpc.normal  = [ 0.0 , 1.0 , 0.0 ]
17 fpc.radius  = trick.attach_units( "M",1.0)
18 fpc.param_name = "steel"
19 fpc.mass_body_name = "vehB"
20 fpc.name = "collision face"
21
22 interactions.vehB_surface.add_facet(fpc)
```

4. See S_define.

   (a) See Modified_data/basic_pair.py

   (b) See Modified_data/stiff_pair.py. Note that both pairs have to be “steel-steel”, since the two facets are defined as being “steel”. Thus, the two sets of contact-pair data have to be in separate files, with only one being included in any given scenario.

5. See S_define, lines 77-88

6. Because there are four runs, I made a common input file at Common_run/input.py. Then, for RUN_Contact, I added the spring-pair parameters defined in basic_pair.py.

```
1 execfile("Common_run/input.py")
2 execfile("Modified_data/basic_pair.py")
```

7. In RUN_no_contact, I ran the same simulation as RUN_contact, but with the contact model de-activated.

```
1 execfile("RUN_contact/input.py")
2 interactions.contact.active = False
```

8. In RUN_stiff, I substituted basic_pair.py for stiff_pair.py

```
1 execfile("Common_run/input.py")
2 execfile("Modified_data/stiff_pair.py")
```

9. In the no-contact scenario, the two facets simply pass through one another. In the contact scenario, *vehicleB* is

brought to rest and rebounds very slowly along the -y-axis, and *vehicle* moves away along the +y-axis.
Note - while momentum appears to be conserved, this does provide a small violation of conservation of energy; the combined kinetic energy post-collision exceeds the pre-collision value.
In the stiff system, the violation of conservation of energy is very pronounced. The two vehicles rebound at very high relative speeds.
The cause of this is the time-step being too large. The two facets pass through the threshold, begin interacting, and the force that is generated is applied over too large of a time-step, causing non-physical velocity changes.

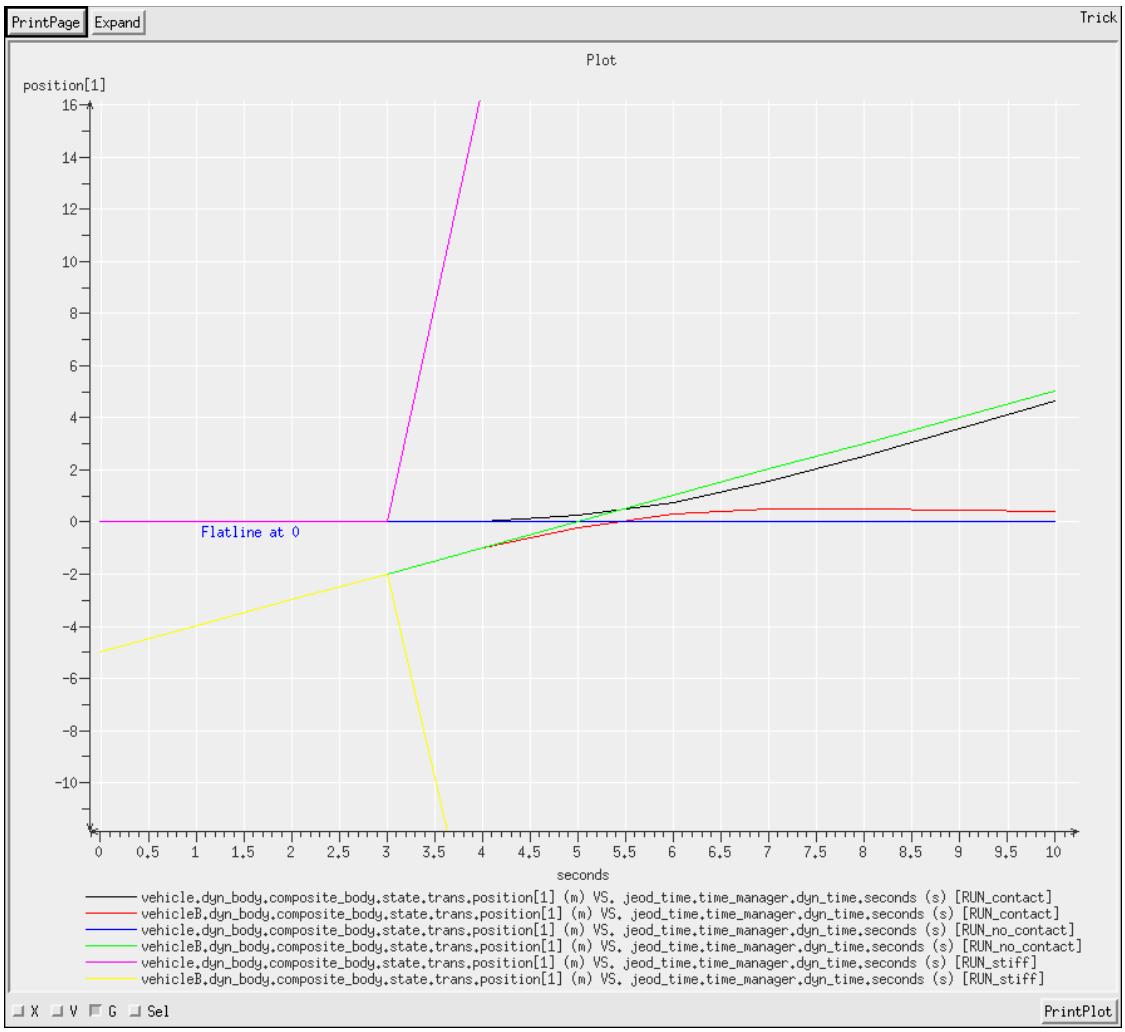
The three plots below show, in order,

(a) the positions of the two vehicles as a function of time,

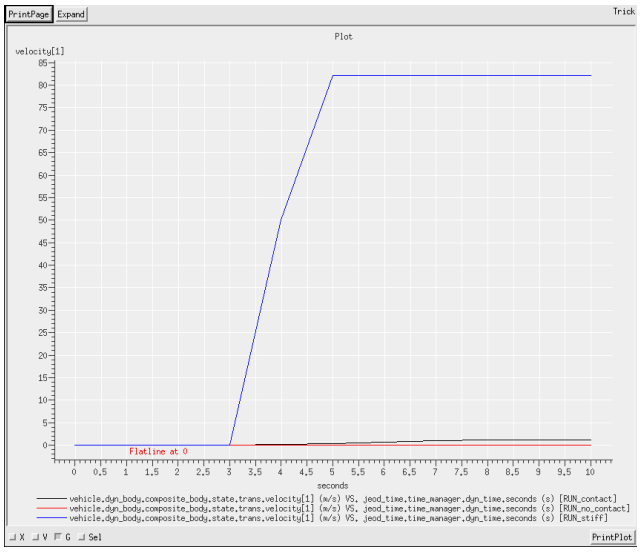
(b) the speed of *vehicle* (along the y-axis*)*

(c) the speed of *vehicle* (along the y-axis*)*, closer view

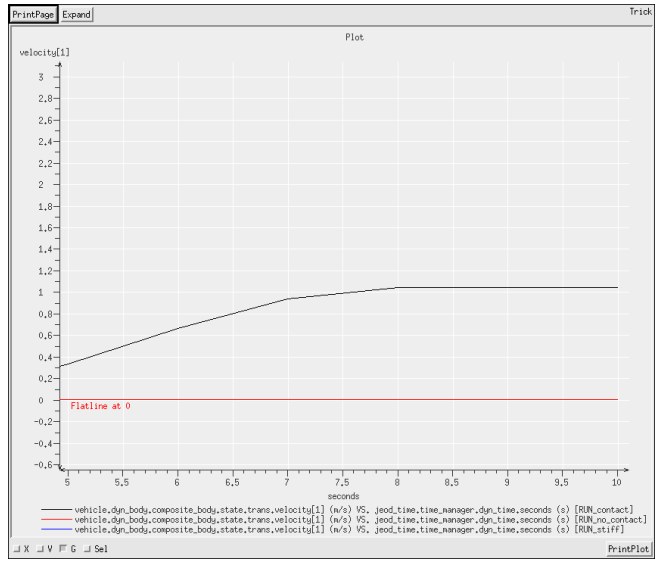
(a)



(b)



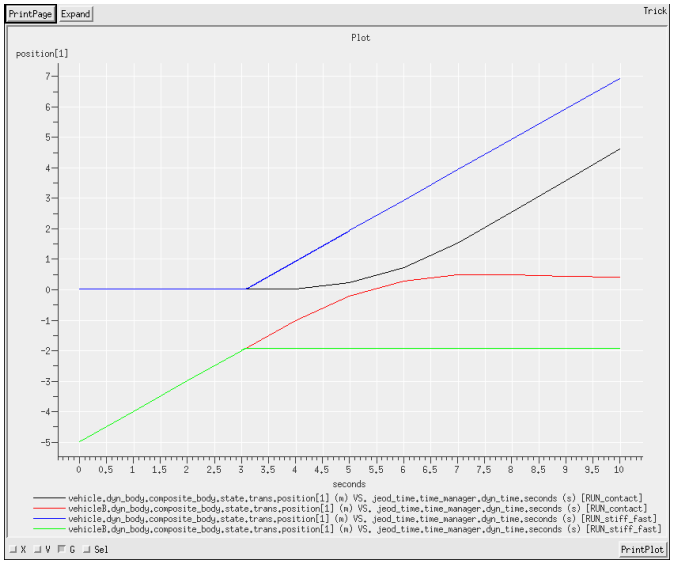
(c)

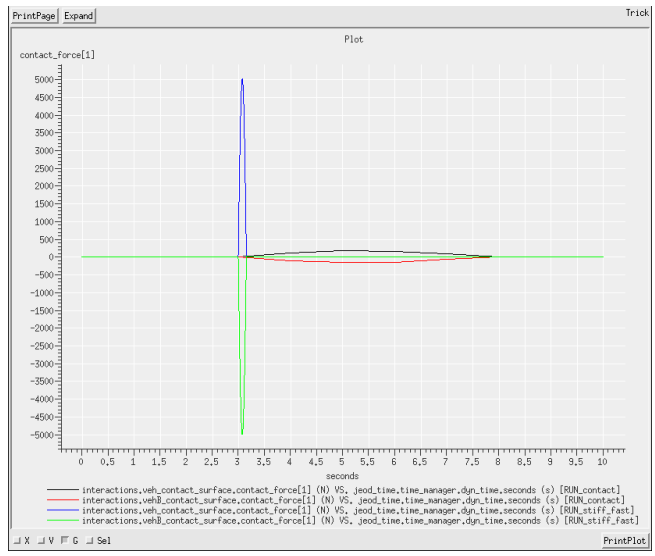


10. The error in RUN_stiff shows at t = 3.0s, so at t=3.0s, make the clock run 1000x faster. Keep this for a couple of seconds (guess, it should be over more quickly than RUN_contact, and that interaction lasted about 3 seconds), then revert back to regular speed. Note that the transitions are specified as sim-time, so 2 seconds of dyn-time is 2000 ticks of sim-time, and the clock reverts at sim-time = 2.0 * 1000 + 3.0. Similarly, the stop time must also be increased by (2000 -2) ticks.

```
trick.add_read(3.0, """
jeod_time.time_manager.dyn_time.scale_factor = 0.001
""")

trick.add_read(2003, """
jeod_time.time_manager.dyn_time.scale_factor = 1.0
""")

trick.sim_services.exec_set_terminate_time(2008.0)
```

11. The plots show position, and the force profile.





Clearly, the stiffer spring responds more forcibly, as would be expected.

Exercise 30 Articulating Surfaces

30.1 Re-orienting an Array

1. I'm using *time_dyntime_only.sm* and *dynamics_init_only.sm* simulation modules.
2. I copied *vehicle_basic.sm* into the S_define, and edited. For the previous interactions exercises, I added a simulation object to add to the vehicle; this time I will add to the vehicle directly.
3. I'm using the offset-orientation attachment-specification (*BodyAttachMatrix*) rather than the mass-point-based attachment *(BodyAttachAligned)*. It is a little easier to configure for this scenario, but either works perfectly well.
4. I added a pair of *FlatPlate* instances, and a *SurfaceModel* to the vehicle simulation-object.
5. I defined a pair of facets in *Modified_data/surface.py*. Assigned the names *vehicle* and *array* to the respective *mass_body_name* values. Note – the names are assigned to the MassBody instances in the input file:

```
RUN_part_1/input.py
 10 vehicle.dyn_body.set_name("vehicle")
 15 vehicle.array.set_name("array")
```

6. See *Modified_data/surface.py*
7. This exercise illustrates the use of multiple *BodyReattach* processes. This is suitable for simulation a small number of discrete rotations. The implementation is found in the *RUN_part_1/input.py* file.
   Four instances of the *BodyReattach* are allocated, then each populated and added to the list of body-actions. Each is initially de-activated for later activation. These body-actions reconfigure the orientation of the array relative to the parent; their effects are NOT cumulative, they do NOT represent a rotation about the specified axis from the current position to some new position.

   NOTE - Consequently, this method does not lend itself to multiple callings of a small angle rotation for generating smooth motion. If that were tried, the array would move to that small angle relative to the parent frame, and stay there.

```
RUN_part_1/input.py
 23 vehicle.reattachments = trick.alloc_type(4,"jeod::BodyReattach")
 24
 25 vehicle.reattachments[0].subject = vehicle.array
 26 vehicle.reattachments[0].offset_pstr_cstr_pstr = [0,0,0]
 27 vehicle.reattachments[0].pstr_cstr.data_source = trick.Orientation.InputEigenRotation
 28 vehicle.reattachments[0].pstr_cstr.eigen_angle = trick.attach_units( "d", 90.0 )
 29 vehicle.reattachments[0].pstr_cstr.eigen_axis  = [1.0, 0.0, 0.0]
 30 vehicle.reattachments[0].active = False
 31 dynamics.dyn_manager.add_body_action( vehicle.reattachments[0] )
 32
 33 vehicle.reattachments[1].subject = vehicle.array
 34 vehicle.reattachments[1].offset_pstr_cstr_pstr = [0,0,0]
etc
```

   Next, at the desired time or event, the appropriate body-action is activated and the Dynamics Manager method *perform_actions* called.

```
RUN_part_1/input.py
 57 trick.add_read(2.0 , """
 58 vehicle.reattachments[0].active = True
 59 dynamics.dyn_manager.perform_actions()
 60 """)
 61
 62 trick.add_read(4.0 , """
etc
```

8. There are four rotations – into positions representing 90 degrees on each of the x-, y-, and z- axes, and 180 degrees on the z-axis.

   (a) Normal remains constant as expected.

(b) The *array* facet normal lies along the array-body x-axis, which starts aligned with the *vehicle* x-axis.

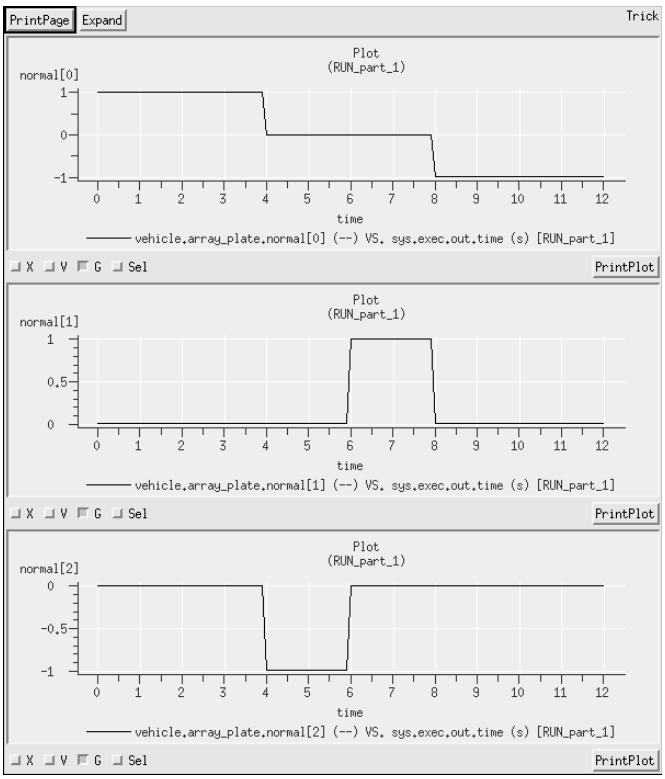
i. T = 2.0 s. Rotating the *array* about the x-axis leaves the *array* facet normal along the *vehicle* x-axis

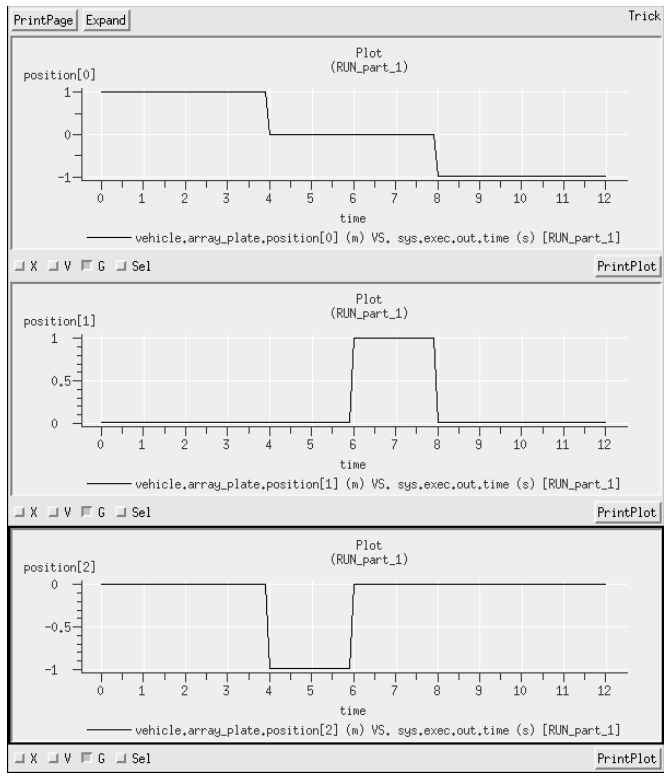
ii. T = 4.0 s. Rotating the *array* through 90 degrees about the y-axis moves the *array* facet normal to be along the *vehicle* -*z*-axis

iii. T = 6.0 s. Rotating the *array* through 90 degrees about the z-axis moves the *array* facet normal to be along the *vehicle* *y*-axis

iv. T = 8.0 s. Rotating the *array* through 180 degrees about the z-axis moves the *array* facet normal to be along the *vehicle* -*x*-axis

9. Because the facet is positioned along the same axis as the facet normal is oriented, the position of the facet should follow the same pattern.





30.2 Verifying the Re-orientation

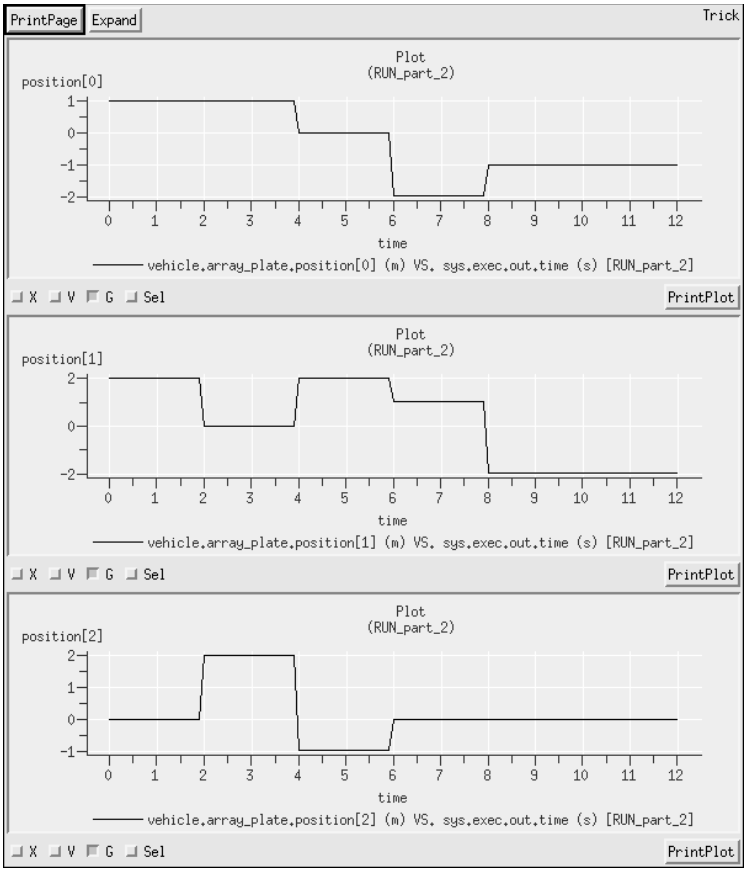
1. See *Modified_data/surface_part2.py* and *Modified_data/vehicle_mass_props_part2.py*
2. Task
3. Orientations are unchanged, but the positions of the array facet are significantly different. See chart below

   i. The vector [1,2,0] rotated 90-degrees about the x-axis becomes [1,0,2]

   ii. The vector [1,2,0] rotated 90-degrees about the y-axis becomes [0,2,-1]

   iii. The vector [1,2,0] rotated 90-degrees about the z-axis becomes [-2,1,0]

   iv. The vector [1,2,0] rotated 180-degrees about the z-axis becomes [-1,-2,0]



Exercise 31 Smooth Rotation, with Radiation Pressure

1. Task
2. In S_define, adding the basic environment and sun simulation objects (lines 15,16)
3. Where is the sun? Solution here is to orient the vehicle with *Sun.inertial*, position the vehicle along the x-axis of *Sun.inertial*, and point the arrays along the -x-axis of the vehicle, which is the -x-axis of *Sun.inertial* (since the two are aligned). See *Modified_data/vehicle_state.py*.
4. Added an interactions simulation-object, based on the one from Exercise 28.
5. In the S_define, there are two new variables – to represent the transformation matrix and position of *array* relative to *vehicle*. A new method, *compute_T_rot* is defined **in the S_define**; this method takes the current time, and produces a time-dependent transformation matrix that represents a rotation about the y-axis. Note that the *BodyAttachMatrix* is still needed to make the initial attachment, but the *BodyReattach* has gone away..

```
S_define
 56    double T_for_rotation[3][3];
 57    double facet_pos[3];
```

```
void compute_T_rot( double time_in)
{
   T_for_rotation[0][0] = std::cos(time_in * 3.14 / 100.0);
   T_for_rotation[0][2] = std::sin(time_in * 3.14 / 100.0);
   T_for_rotation[2][0] = -std::sin(time_in * 3.14 / 100.0);
   T_for_rotation[2][2] = std::cos(time_in * 3.14 / 100.0);
}
```

Instead of calling the body-action for the reattach process, the method is called directly in the S_define. First, the transformation-matrix is re-computed, then that is fed to the reattach method call before the surface articulation is updated.

```
S_define
   (DYNAMICS, "environment") compute_T_rot(dyn_time);
   (DYNAMICS, "environment") array.reattach(facet_pos,T_for_rotation);
   (DYNAMICS, "environment") surface.update_articulation();
```

6. Two plots shown. The first provides 3 plots:

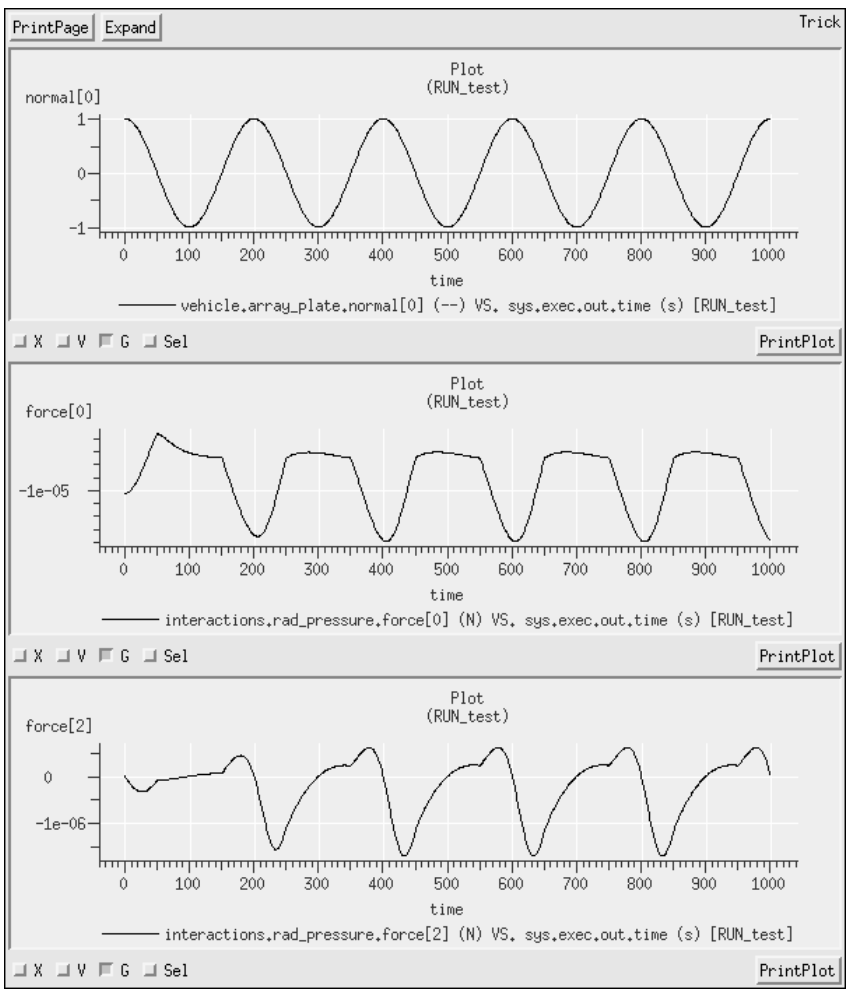
   (a) the x-component of the normal vector; this provides the benchmark for interpreting the orientation of the facets.

   (b) the x-component of the force on the vehicle

   (c) the z-component of the force on the vehicle.

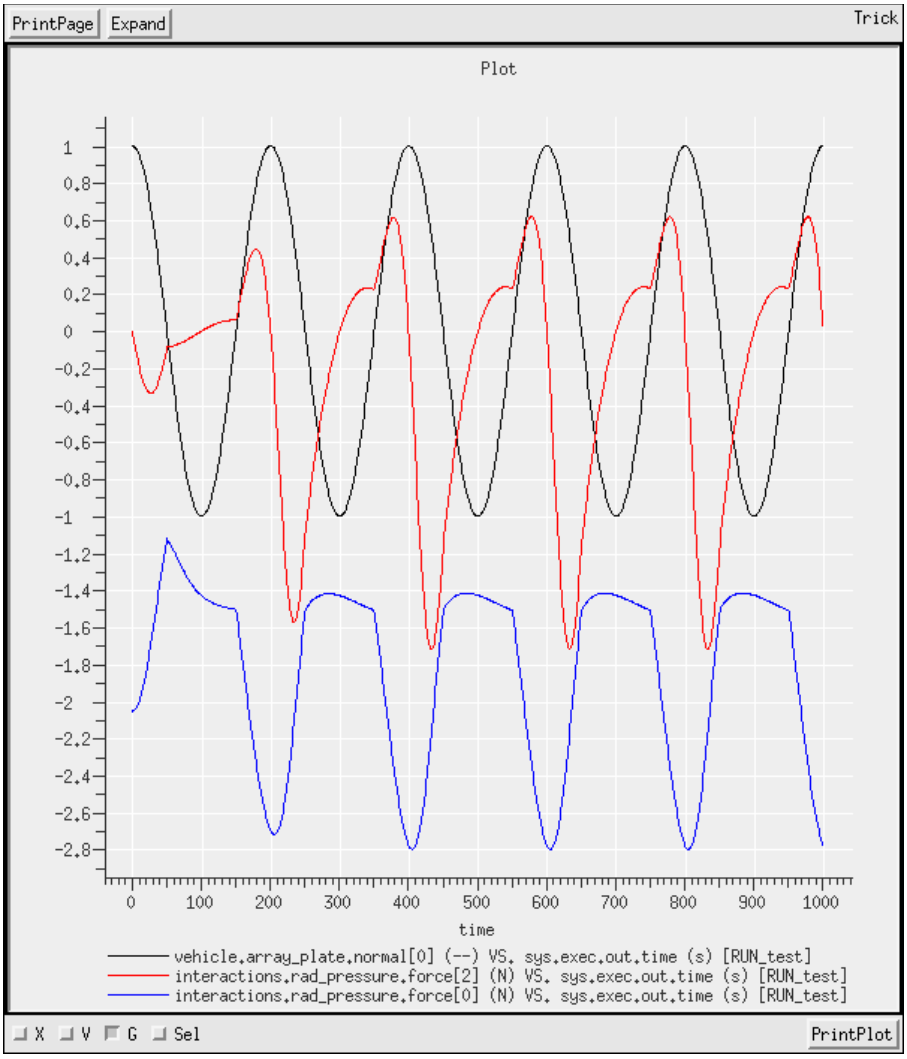
   (note that the y-component of force was zero)

   The second graph scales the values and co-plots the three plots from the first graph to illustrate more clearly how the axes react at particular times.



On the x-axis, there is clearly a strong force in the -x direction that grows as the facet rotates into direct sunlight (normal[0] = 1), and then decays as it rotates back out again. As the array facet moves into shadow, only the vehicle facet is left, which provides a constant force. Notice that the total x-component of force slightly lags the illumination, and is not constant while the array facet is shadowed; these are consequences of the thermal effects; the strongest force along the -x-axis will occur slightly after direct illumination, as the facet continues to warm through the direct-illumination phase. After the array rotates into shadow, it will continue to exert a force but in the +x direction; that force will vary sinusoidally with angular position, and decay with decreasing temperature.

On the z-axis, the only force will be from the *array* facet. There is a rapid jump in the force as the facet moves into sunlight, then falls to zero as the facet reaches direct-illumination. As the facet continues past direct-illumination, the sign of the z-force changes as the normal picks up a +z component (so the force has a -z component). The marked asymmetry between the +z and -z axes is known as the Yarkovsky effect, a consequence of the temperature differential between the two twilight regions of the vehicle.

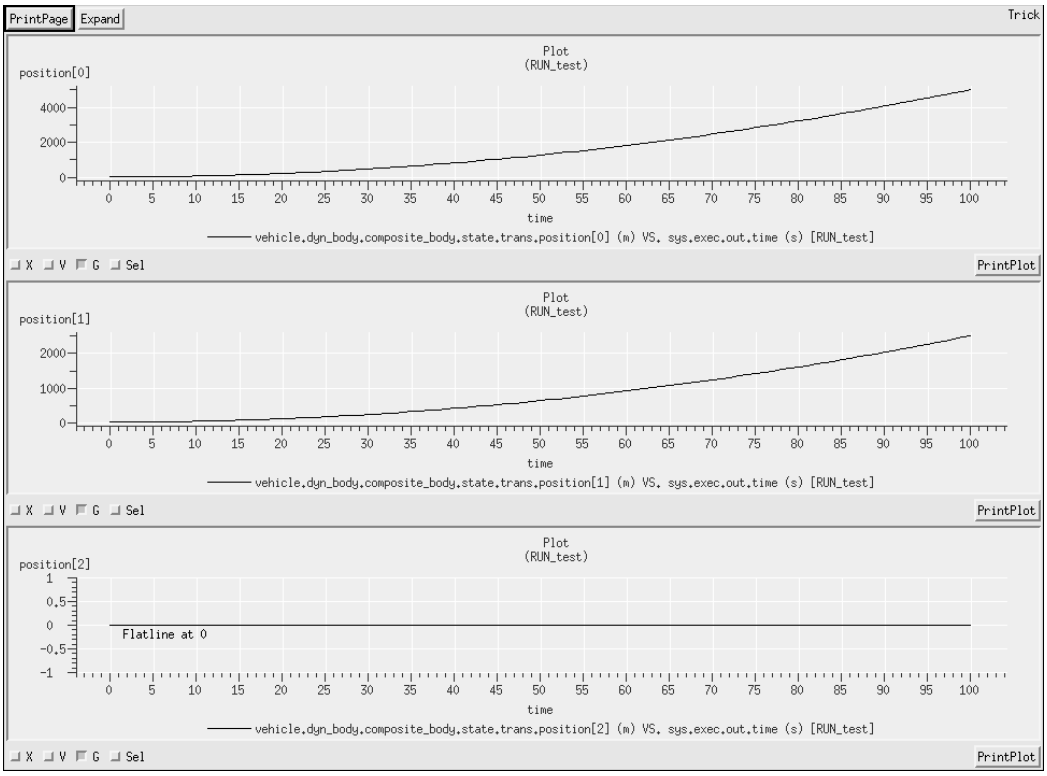


Exercise 32 Simple Simulation of Vehicle with Thrusters

1. See S_define (lines 8-13), and Modified_data files.
2. Added a simulation-object *effector* to include the force from a thruster. See S_define, lines 16-25. Also collected that force, and applied it to the vehicle (lines 27-28). The force is defined in *Modified_data/vehicle_mass_props.py*

```
11 effector.thruster.force = [10,5,0]
```

3. As expected, a quadratic curve in time on the forcing axes with a magnitude proportional to the force.



Exercise 33 Revisiting an Orbital-Elements Initialized Simulation; Output Orbital-Elements

WARNING – because this simulation uses non-spherical gravity, it will look for *Earth.pfix*. It is generally assumed that *Earth.pfix* has a state relative to some inertial frame, such as *EMBary.inertial*, but this is a *SinglePlanet* simulation, so it does not. Consequently, there will be a warning that *Earth.pfix* has no owner. Such warnings are intended to alert the user that the simulation is not configured as expected. In this case, we already know that, so this warning can be ignored.

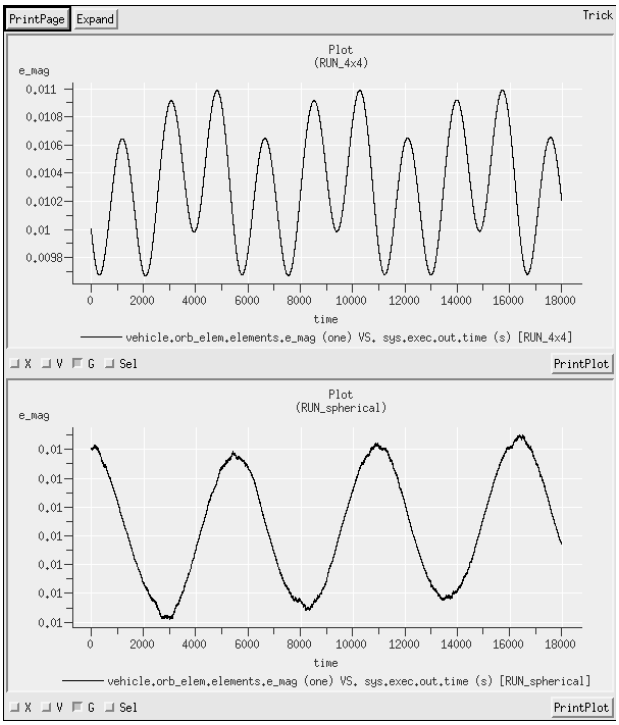
1. Task
2. Switch *earth_basic.sm* for *earth_GGM02C.sm*
3. Added *OrbElemDerivedState* to the S_define

```
21 ##include "dynamics/derived_state/include/orb_elem_derived_state.hh"
...
29    jeod::OrbElemDerivedState orb_elem;
```

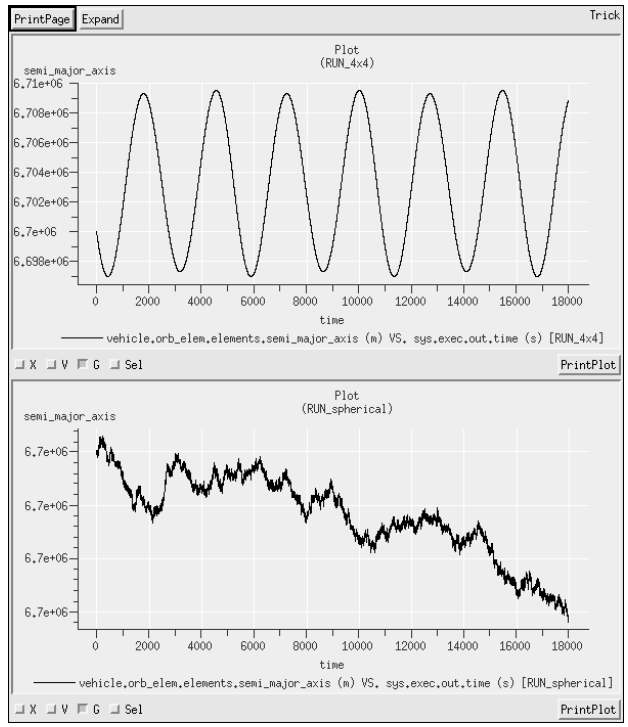
4. Edited Modified_data/vehicle_grav_controls.py to provide a 4x4 field. Used *RUN_4x4* as the principle input file; in *RUN_spherical*, the *vehicle.grav_control.spherical* value is reset to True, but otherwise uses the *RUN_4x4/input.py* file.
5. The eccentricity, semi-major-axis and inclination should remain approximately constant, while the true-anomaly should increase approximately linearly (eccentricity is small) from 0 to 2π, then return to 0 producing a saw-tooth pattern.
   Be aware that orbital elements are based on Keplerian orbits with a point-mass focus. Where this assumption is violated, the vehicle will not fly a perfect ellipse, and the calculation of orbital elements will be only approximate.

   The first three graphs below ((a) – (c)) show the eccentricity, sma and inclination for both runs, with the 4x4 gravity in the upper pane, and the spherical case in the lower pane. The 4th (d) shows the true anomaly of both in the upper pane, with the difference in the lower pane.

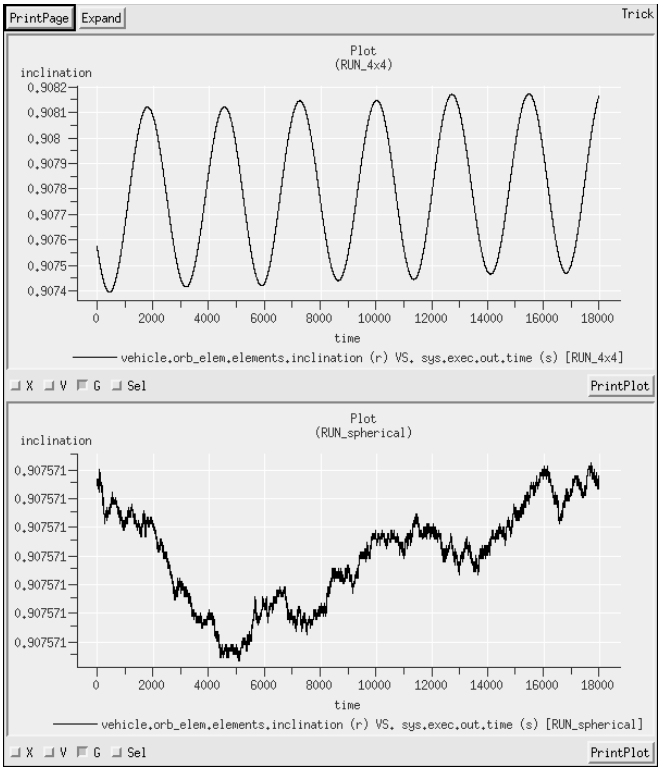
(a)



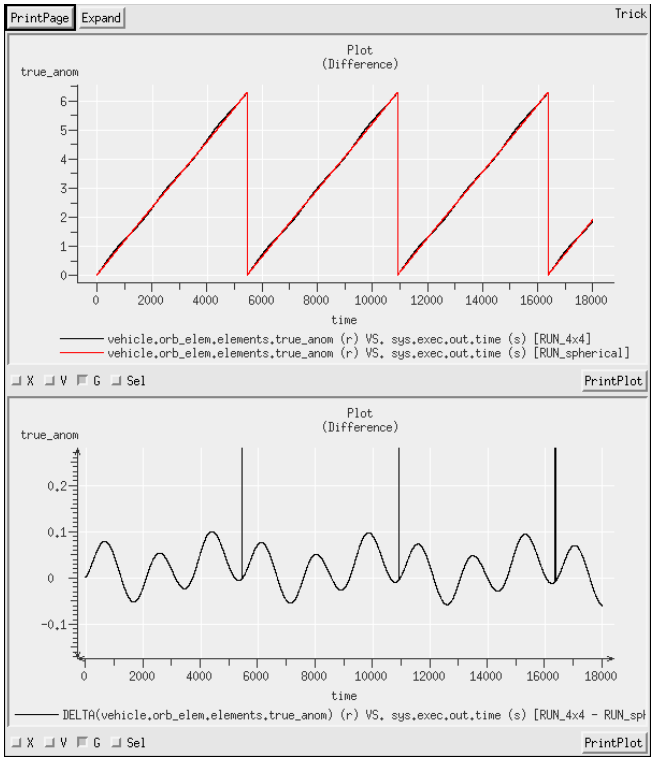
(b)



(c)



(d)



The elements associated with the vehicle in spherical gravity remain very close to constant, while those for the 4x4 case vary widely. There is an important lesson to be learned here, that the calculation of orbital elements is very susceptible to perturbations brought on by non-spherical gravitational fields. It is generally unreliable to make a single calculation of orbital elements when the gravity is non-spherical, or when there are other perturbing effects that make the vehicle fly away from the perfect ellipse.

Exercise 34 Derived States Exercise - Orbital Elements and Planet Fixed States

1. Task
2. In Exercise 33, the orbital-elements output and orbital-elements-based state initialization (*OrbElemDerivedState* and *DynBodyInitOrbit*) were added by creating a new simulation-object for the convenience of not having to edit the pre-existing vehicle simulation-object. However, to follow the same design in this exercise, I would have to create 2 instances of the *OrbElemDerivedState* instance (for 2 vehicles) in the new *state* sim-object, along with 2 instances of *DynBodyIniitLvlhRotState* (for 2 vehicles) and an instance of *DynBodyInitLvlhTransState* (for vehicleB), and that would just be confusing. So the *state* sim-object is abandoned, and the contents rolled back into the vehicle simulation-object.

   Note the trade-off here. I now have an unused *DynBodyInitOrbit* instance on vehicleB, and an unused *DynBodyInitLvlhTransState* instance on vehicleA.

   I changed the name from of the Modified-data file from *vehicle_state_orbelm.py* to *vehicle_state.py* because it will now contain not only the initial orbital elements for vehicleA, but also the attitude for vehicleA and the complete state for vehicleB; these additions have no relation to orbital elements, and the file name should reflect that. See *Modified_data/vehicle_state.py* for specific content, and solutions to Exercise 24 for explanation of initializing relative to LVLH; nothing new here.

   These data lead to the choice of initializing the translational state of *vehicleB* in the LVLH frame of *vehicleA*.

```
26 vehicleB.lvlh_trans_init.ref_body_name  = “vehicleA”
```

   Note that, in this case, the simulation requirements do not specify any need for an LVLH representation beyond initialization. So I am using the Lvlh-initialization option that creates and manages its temporary LvlhFrame instance internally.

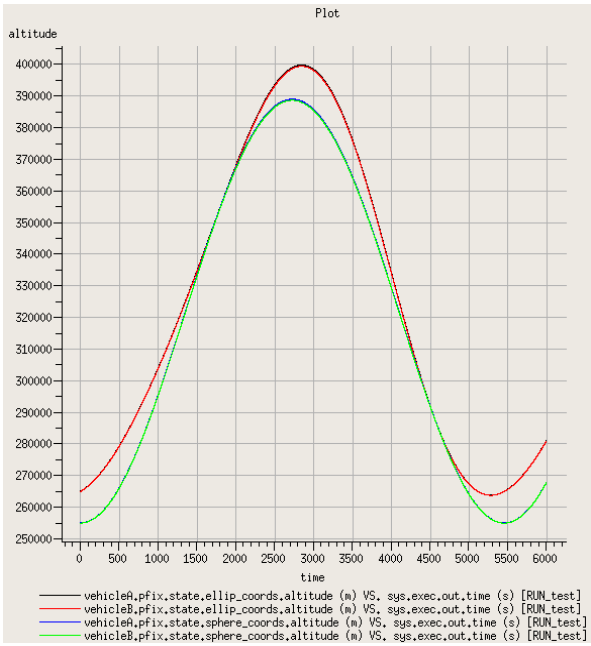
   Use *DynBodyInitLvlhRotState* for the rotational state. The specification of which vehicle to use for LVLH defaults to self-referencing for the rotational initialization. Hence, it is not necessary to provide a *ref_body_name* for this scenario.

3. Changes to the S_define; adding *OrbElemDerivedState* and *PlanetaryDerivedState* to the vehicle-sim-object so that they get added to both vehicles.

   Note – the single instance of *PlanetaryDerivedState* (per vehicle) provides both spherical and elliptical representations (per vehicle).

```
30 ##include "dynamics/derived_state/include/orb_elem_derived_state.hh"
31 ##include "dynamics/derived_state/include/planetary_derived_state.hh"
...
44    jeod::OrbElemDerivedState orb_elem;
45    jeod::PlanetaryDerivedState pfix;
...
52      P_DYN  ("initialization") orb_elem.initialize ( dyn_body,
53                                                      dyn_manager);
54      P_DYN  ("initialization") pfix.initialize ( dyn_body,
55                                                 dyn_manager);
...
57      (DYNAMICS, "environment") orb_elem.update ( );
58      (DYNAMICS, "environment") pfix.update ( );
```

4. The eccentricity and semi-major axes are distinctly different between the two vehicles and are each approximately constant in time (this is with spherical gravity, so the orbit should be Keplerian, with regular orbital elements). The altitude plot shows more variation between spherical and elliptical representations (about 10 km) than between the two vehicles (50 m – 300 m). The longitude of *vehicleB* starts to creep ahead of that of *vehicleA*, as expected (it has smaller orbital period).


Plot
altitude
400000
390000
380000
370000
360000
350000
340000
330000
320000
310000
300000
290000
280000
270000
260000
250000
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
time
vehicleA.pfix.state.ellip_coords.altitude (m) VS. sys.exec.out.time (s) [RUN_test]
vehicleB.pfix.state.ellip_coords.altitude (m) VS. sys.exec.out.time (s) [RUN_test]
vehicleA.pfix.state.sphere_coords.altitude (m) VS. sys.exec.out.time (s) [RUN_test]
vehicleB.pfix.state.sphere_coords.altitude (m) VS. sys.exec.out.time (s) [RUN_test]

Exercise 35 Derived States Exercise – Relative State

1. This looks a lot like Exercise 34, but with different initial orbital elements. Copied the contents of SIM_34 as a basis.

2. Edited *S_define* to remove the orbital-elements and planet-fixed state, and add the new Relative-state. This was added to its own simulation-object because it does not really belong with either

```
~~30 ##include "dynamics/derived_state/include/orb_elem_derived_state.hh"~~
~~31 ##include "dynamics/derived_state/include/planetary_derived_state.hh"~~
~~...~~
~~44    jeod::OrbElemDerivedState orb_elem;~~
~~45    jeod::PlanetaryDerivedState pfix;~~
~~...~~
~~52      P_DYN  ("initialization") orb_elem.initialize ( dyn_body,~~
~~53                                                      dyn_manager);~~
~~54      P_DYN  ("initialization") pfix.initialize ( dyn_body,~~
~~55                                                    dyn_manager);~~
~~56~~
~~57      (DYNAMICS, "environment") orb_elem.update ( );~~
~~58      (DYNAMICS, "environment") pfix.update ( );~~

61 ##include "dynamics/derived_state/include/relative_derived_state.hh"
...
67    jeod::RelativeDerivedState  vehicleB_wrt_vehicleA_bodyA;
...
76      P_DYN  ("initialization") vehicleB_wrt_vehicleA_bodyA.initialize (
77                                                  vehB_body,
78                                                  dyn_manager);
79      (DYNAMICS, "environment") vehicleB_wrt_vehicleA_bodyA.update ( );
```

   Edited *Modified_data/derived_state.py* to remove the orbital-element and planet-fixed state configurations and instead setup the two LVLH frames.

```
Modified_data/derived_state.py
 1 rel_state.vehicleB_wrt_vehicleA_bodyA.subject_frame_name = "vehicleB.composite_body"
 2 rel_state.vehicleB_wrt_vehicleA_bodyA.target_frame_name = "vehicleA.composite_body"
 3 rel_state.vehicleB_wrt_vehicleA_bodyA.direction_sense   =
                                              trick.RelativeDerivedState.ComputeSubjectStateinTarget
```

3. Edited *Modified_data/vehicle_state.py* to change the orbital elements

   There are only 3 pieces of data – semi-major-axis, eccentricity (0.0, circular), and inclination (0.0, equatorial). Note that for an orbit defined relative to the equator, the orbital elements should be set relative to the ECEF frame *(Earth.pfix)* rather than ECI *(Earth.inertial);* to express with respect to *Earth.inertial* we would need to know the inclination of the equator. However, in this case, we are in *SinglePlanet* mode, so ECEF and ECI are equivalent.
   There are two options to initialize from orbital elements that include these three data points. This solution arbitrarily uses the *SmaEccIncAscnodeArgperTimeperi* option. The other pieces of data are not important (there is no ascending node with zero inclination, and there is no periapsis in a circular orbit)

4. Task

5. As expected, the position and velocity vary sinusoidally on x- and y-axes, while remaining flat at zero on the z-axis (polar axis). The vehicles start on the +x axis, with velocity on the +y axis (note – this was not specified because we did not define the initial orientation of the orbit or the initial position of the vehicle on the orbit but is default behavior). The body axes are oriented with respect to the LVLH frame, so we have a mapping -

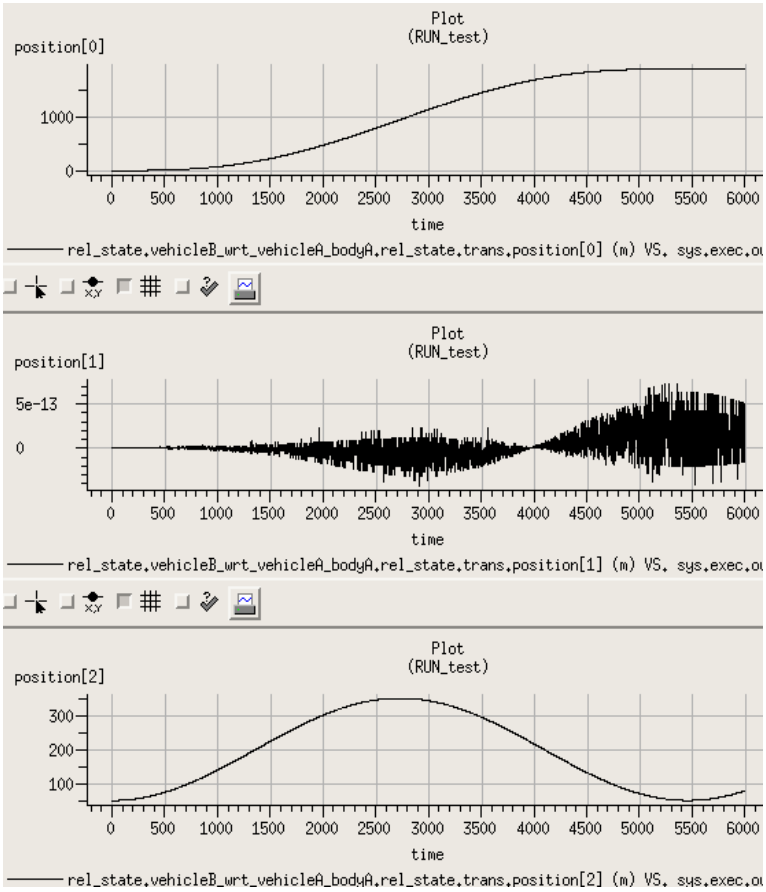
   inertial x-axis <-> body -z-axis (initial position)

   inertial y-axis <-> body x-axis (initial velocity)

inertial z-axis <-> body -y-axis (complete the frame)

The orientation, as monitored through the transformation matrix, matches these predictions.

6. *VehicleB* starts to creep ahead of *vehicleA*, as it did in Exercise 34. This is because *vehicleB* is at a lower altitude. Because *vehicleB* starts at a lower altitude and the same velocity as *vehicleA* (which is in a circular orbit), *vehicleB* must be in an eccentric orbit, and starts at apoapsis. As it drops towards periapsis, the relative z-position (the body frame is oriented with the LVLH frame, so z-axis points "down") increases, then decreases again as *vehicleB* climbs back to apoapsis.



Exercise 36 Derived States Exercise - Define a LVLH Reference Frame

1. This looks a lot like Exercise 34, but with different initial orbital elements. Copied the contents of SIM_34 as a basis.
2. Edited *S_define* to remove the orbital-elements and planet-fixed state, and add the new LVLH frame to each vehicle.

```
~~30 ##include "dynamics/derived_state/include/orb_elem_derived_state.hh"~~
~~31 ##include "dynamics/derived_state/include/planetary_derived_state.hh"~~
~~...~~
~~44    OrbElemDerivedState orb_elem;~~
~~45    PlanetaryDerivedState pfix;~~
~~...~~
~~52      P_DYN  ("initialization") orb_elem.initialize ( dyn_body,~~
~~53                                                      dyn_manager);~~
~~54      P_DYN  ("initialization") pfix.initialize ( dyn_body,~~
~~55                                                    dyn_manager);~~
~~56~~
~~57      (DYNAMICS, "environment") orb_elem.update ( );~~
~~58      (DYNAMICS, "environment") pfix.update ( );~~

30 ##include "utils/lvlh_frame/include/lvlh_frame.hh"
...
43    jeod::LvlhFrame        lvlh_frame;
...
50      P_DYN  ("initialization") lvlh_frame.initialize ( dyn_manager);
51
52      (DYNAMICS, "environment") lvlh_frame.update ( );
```

Edited *Modified_data/derived_state.py* to remove the orbital-element and planet-fixed state configurations and instead setup the two LVLH frames.

```
Modified_data/derived_state.py
 1 vehicleA.lvlh_frame.subject_name = "vehicleA.composite_body"
 2 vehicleA.lvlh_frame.planet_name = "Earth"
 3
 4 vehicleB.lvlh_frame.subject_name = "vehicleB.composite_body"
 5 vehicleB.lvlh_frame.planet_name = "Earth"
```

3. Copied *Modified_data/vehicle_state.py* from Exercise 35. Because I have the *LvlhFrame* instance already available, I will use that instead of having the *DynBodyInitLvlh*State* create a temporary one.
4. Task
5. The results match perfectly. The presence of the LVLH frame has absolutely no effect on the vehicle states, as should be expected.

Exercise 37 Derived States Exercise - Relative LVLH State

1. Task
2. Typically, when there are multiple relative states between vehicles, it is conventional to put them in into their own dedicated simulation-object that appears after the vehicle sim-objects in the S_define. This time, I wanted to try putting these into the vehicle sim-objects to demonstrate that it could be done, and to demonstrate the difficulties associated with doing so.

   To get over the dependency issues on the status of the *LvlhFrame* (see discussion in the course text), I had to make two rather kludgy modifications:

   (a) I moved the initialization of the LVLH state to a *P_GRAV* priority level to ensure that it initialized AFTER both instances of LvlhFrame, which have *P_DYN* priority.

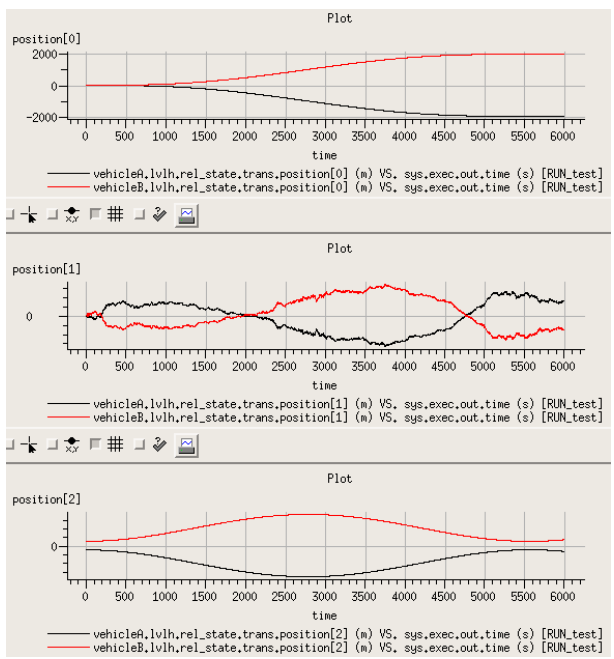
(b) I gave the environment-class jobs that update the LvlhFrame instances a priority to ensure that these jobs complete before the LVLH-state updates are called.

Note that this tactic could have unintended consequences. For a large simulation, I would strongly suggest creating a relative-state simulation object, as was done in Exercise 35.

```
S_define
    31 ##include "dynamics/derived_state/include/lvlh_relative_derived_state.hh"
    ...
    45    jeod::LvlhRelativeDerivedState lvlh;
    ...
    54      P_GRAV ("initialization") lvlh.initialize ( dyn_body,
    55                                                  dyn_manager);
    56
    57      P_DYN (DYNAMICS, "environment") lvlh_frame.update ( );
    58      (DYNAMICS, "environment") lvlh.update ( );
```

```
Modified_data/derived_state.py
     7 vehicleA.lvlh.name               = "vehicleA_in_vehicleB_LVLH"
     8 vehicleA.lvlh.subject_frame_name = "vehicleA.composite_body"
     9 vehicleA.lvlh.target_frame_name  = "vehicleB.composite_body.Earth.lvlh"
    10 vehicleA.lvlh.lvlh_type          = trick.LvlhType.Rectilinear
    11
    12 vehicleB.lvlh.name               = "vehicleB_in_vehicleA_LVLH"
    13 vehicleB.lvlh.subject_frame_name = "vehicleB.composite_body"
    14 vehicleB.lvlh.target_frame_name  = "vehicleA.composite_body.Earth.lvlh"
    15 vehicleB.lvlh.lvlh_type          = trick.LvlhType.Rectilinear
```

3. The two states exhibit a nice symmetry.



4. Created RUN_curvi. The input file for this run executes the RUN_test input file, and changes the LVLH-frame assignment to CircularCurvilinear:

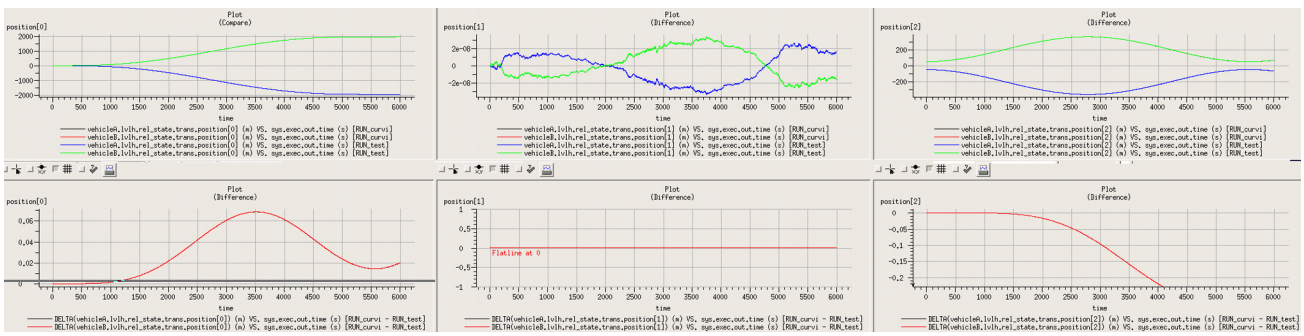
```
vehicleA.lvlh.lvlh_type = trick.LvlhType.CircularCurvilinear
vehicleB.lvlh.lvlh_type = trick.LvlhType.CircularCurvilinear
```

5. Small differences. More significant differences will be observed if the vehicles are separated by a larger distance.

Note, from geometry, the following rules should apply for the mutual relative states of two vehicles with one at a higher-altitude than the other:

- magnitude of the x-values:
  - The rectilinear value of the high-vehicle relative to the low vehicle should be not less than that of the low-vehicle relative to the high-vehicle.
  - The curvilinear value of the low-vehicle relative to the high-vehicle should be not less than the rectilinear value of the same.
  - The curvilinear value of the high-vehicle relative to the low-vehicle should be not greater than the rectilinear value of the same.
- magnitude of the z-values:
  - The rectilinear value of the high-vehicle relative to the low vehicle should be not greater than that of the low-vehicle relative to the high-vehicle.
  - The curvilinear value of the low-vehicle relative to the high-vehicle should be not greater than the rectilinear value of the same.
  - The curvilinear value of the high-vehicle relative to the low-vehicle should be not less than the rectilinear value of the same.

6.



Exercise 38 Solar Beta

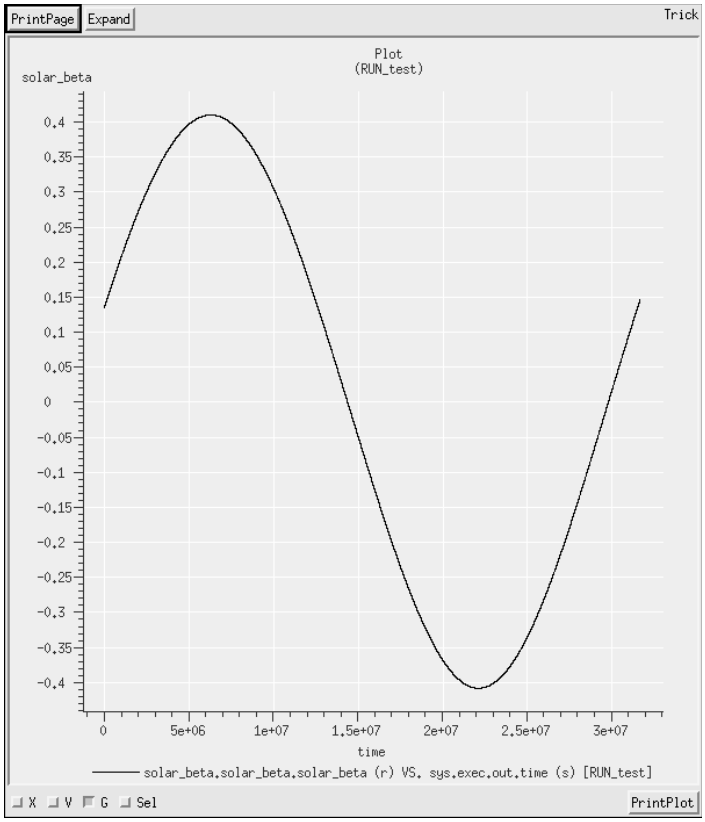
1. Including Sun and Earth means also including Moon; we have seen this requirement before. However, I do not add lunar gravity, so while Moon is present in the sim, it has no effect. See *Modified_Data/vehicle_grav_controls.py* for addition of Earth and Sun gravity.

   I borrowed the mass-properties and elements of the state definition from Exercise 34.

   Operating with multiple planets requires the Ephemerides model, which means an enhanced time simulation-module and the enhanced S_overrides.mk file (see, e.g. SIM_28)

2. Defined a new simulation-object, *state* that includes the solar-beta model and the orbital initialization class.

3. To make the simulation run a little faster, I increased the integration step size (DYNAMICS) to 10.0 seconds (precision is not important here), and the logging interval to 1000.0 seconds.


PrintPage
Expand
Trick
Plot
(RUN_test)
solar_beta
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2
-0.25
-0.3
-0.35
-0.4
0
5e+06
1e+07
1.5e+07
2e+07
2.5e+07
3e+07
time
solar_beta.solar_beta.solar_beta (r) VS. sys.exec.out.time (s) [RUN_test]
X
V
G
Sel
PrintPlot

Exercise 39 Managing Relative States with the Relkin Manager

1. Task
2. See *Modified_data/vehicle_mass_props.py: 13-19*
3. See *Modified_data/vehicle_mass_props.py: 8-10 and Modified_data/vehicle_state.py:11-17, 25-31*
4. See *Modified_data/vehicle_state.py:1-9*
5. In the S_define:

```
 67 ##include "dynamics/rel_kin/include/relative_kinematics.hh"
...
 77    jeod::RelativeKinematics         relkin;
```

6. In the S_define, add the two existing states to *relkin.*

```
 97      P_DYN  ("initialization") relkin.add_relstate(pointA_wrt_bodyB_in_lvlh_bodyB);
 98      P_DYN  ("initialization") relkin.add_relstate(pointB_wrt_bodyB_in_lvlh_bodyB );
```

   (a) In the S_define instantiate, initialize, and add to *relkin*,

```
 76    jeod::RelativeDerivedState  pointA_wrt_pointB_in_strucB;
...
 94      P_DYN  ("initialization") pointA_wrt_pointB_in_strucB.initialize (
 95                                                  veh_A_body,
 96                                                  dyn_manager);
...
 99      P_DYN  ("initialization") relkin.add_relstate(pointA_wrt_pointB_in_strucB );
```

   Note that the LVLH frame of vehicle B is not a vehicle frame, so the subject MUST be vehicle A.

   In Modified_data, set the frame-names and the direction.

```
Modified_data/derived_state.py
  7 relative.pointA_wrt_bodyB_in_lvlh_bodyB.subject_frame_name = "vehicleA.point"
  8 relative.pointA_wrt_bodyB_in_lvlh_bodyB.target_frame_name = "vehicleB.composite_body.Earth.lvlh"
  9 relative.pointA_wrt_bodyB_in_lvlh_bodyB.name = "ptA_wrt_vehB_lvlh"
 10 relative.pointA_wrt_bodyB_in_lvlh_bodyB.direction_sense =
trick.RelativeDerivedState.ComputeSubjectStateinTarget
```

   Note the *subject_frame_name* is a concatenation of the vehicle name *(vehicleA*, from the input file) and the *MassPointInit* name *(point*, from *Modified_data/vehicle_mass_props.py)*

   (b) Similar process to part (a), using the same files. Note that this ins only possible if the mass-point is oriented with the structural axes. It is not possible, in general, to obtain the relative state of one frame relative to a second, expressed in a third. If the mass point were not aligned with structural, there would be 2 choices:

      i. post-process the state to transform by the point-to-structure transformation

      ii. Add another MassPoint at the same location that *is* aligned with the structure-frame.

7. Single call to *update_all.*

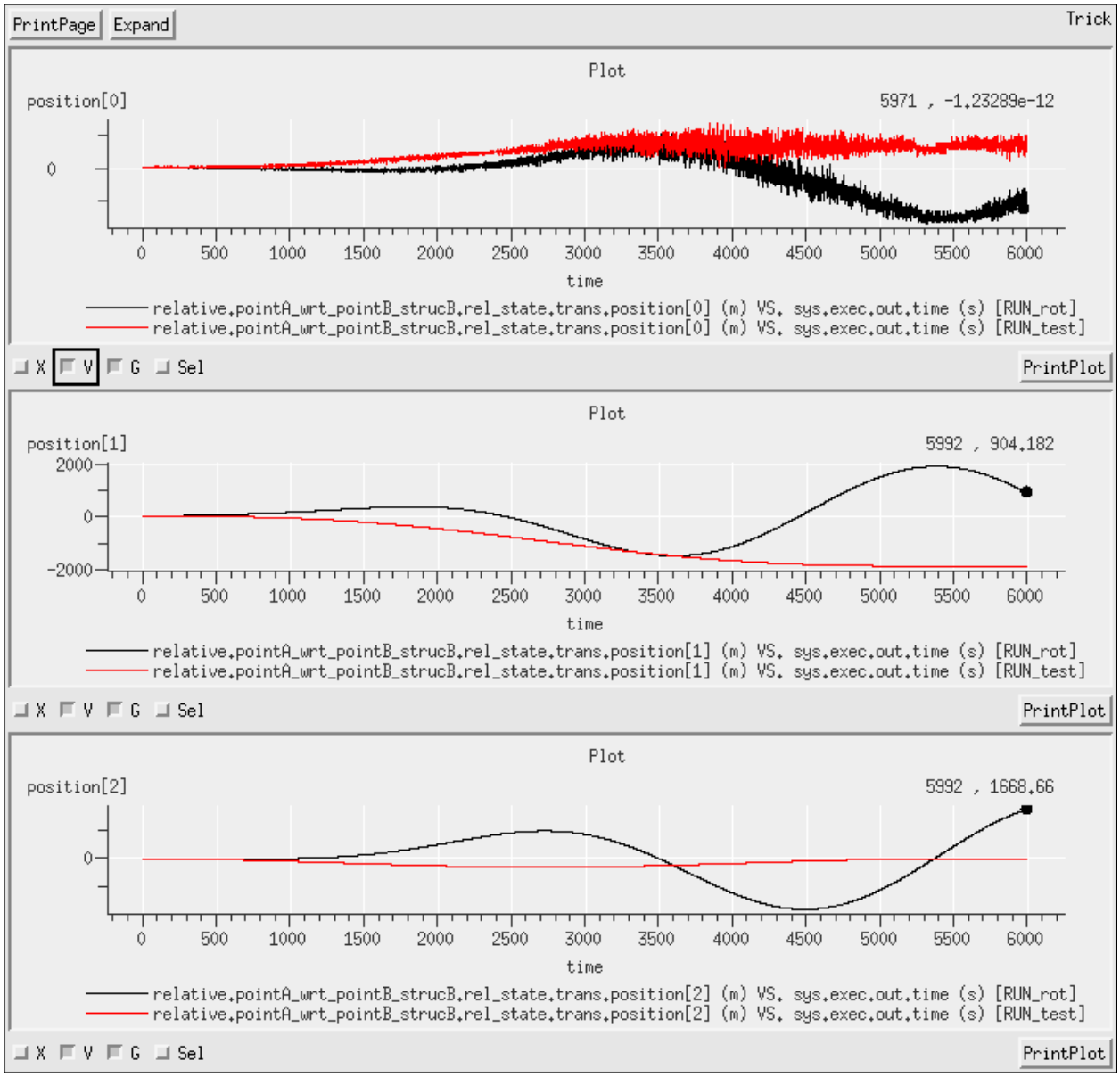
```
101      (DYNAMICS, "environment") relkin.update_all ( );
```

   Results show:

- LVLH x-axis (structural y-axis): *vehicleB* pulls ahead of *vehicleA* by ~2000m after one orbit.
- LVLH y-axis (structural -x axis): this is the axis on which the points lie. The CM – to – CM and mass-point – to – mass-point values remain at 0.0m. The mass-point relative to CM value has a value -1.0m
- LVLH z-axis (structural z-axis): all states show similar results

8. Used *RUN_test*/*input*.py as the baseline input file, and changed only the rotational angular velocity of vehicle B in *RUN_rot*/*input*.py.

The relative states *pointA-with-respect-to-vehicle-B* and *vehicle-A-with-respect-to-vehicle-B* show no changes. With the additional rotation, the body frame is no longer aligned with LVLH, rotating in the x-z plane (equivalently, structural frame rotates relative to LVLH in the y-z plane in the structural frame). Because the additional rotation is on the same axis as the mass-point, the mass-point itself shows no additional motion; the difference is purely a result of the rotating reference frames. This illustrates the necessity of fully understanding the representation of the state outputs when interpreting data:



Note that the position on the x-axis (that is, structural x-axis, LVLH -y-axis) remains at zero, because both mass-points lie at the same position on this axis.

Exercise 40 Managing Relative States with the Relkin Manager, part II

1. Task
2. States (g-i) require new LVLH frames centered on the structure-points.

```
41    jeod::LvlhFrame        lvlh_frame_struc;
...
49      P_ENV  ("initialization") lvlh_frame_struc.initialize ( dyn_manager);
...
53      (DYNAMICS, "environment") lvlh_frame_struc.update ( );
```

   These new frames are initialized in *Modified_data/derived_states.py* in the same manner as *lvlh_frame* was initialized.

   The other states are added as before, see:

   a) S_define, lines 80-85, 107-124, 130-135 and

   b) Modified_data/derived_states.py lines 27-68

3. See *Modified_data/vehicle_state.py*

```
22   vehicleB.lvlh_trans_init.position = [-10000.0, 0.0, 5000.0]
```

4. See *Modified_data/vehicle_mass_props.py*

```
4   VEH.mass_init.properties.position = [0, 5.0, 0]
```

5.

   a) Because the body frame is initialized aligned with and rotating with LVLH, the body-frame of vehicle A and the LVLH frame co-located with it will be aligned for all time. States (d) and (f) should be identical.

   b) There is a 5 meter offset that is visible between the two states on the x-axis. There is also a very small difference on the z-axis.

   c) Substantial difference between the rectilinear and curvilinear representations of the same relative state, as expected.

Exercise 41 Multiple Integration Groups

1.

```
S_define
#define DYNAMICS      1.0

#include "sim_objects/default_trick_sys.sm"
#include "jeod_sys.sm"
#include "default_priority_settings.sm"
#include "integ_loop.sm"

#include "time_dyntime_only.sm"
#include "dynamics_multi_group.sm"
#include "environment_sans_de405.sm"
#include "earth_basic.sm"
#include "vehicle_basic.sm"
VehicleSimObject vehicle2(dynamics.dyn_manager);
VehicleSimObject vehicle3(dynamics.dyn_manager);
```

Note – the definition of DYNAMICS is still necessary because it is used in defining the calling rates of the time-update and the ephemeris-update. However, the integration will also update the time-keeping (at the rate equal to the fastest of the integration groups).

Because all three vehicles have the same data, I created a single method to handle the configuration and called it on all three vehicles, using the vehicle and vehicle-name as arguments. See *Modified_data/vehicle_state.py* for the function definition, called from the input file:

```
RUN_test/input.py
execfile("Modified_data/vehicle_state.py")
set_veh_state(vehicle,  "veh1")
set_veh_state(vehicle2, "veh2")
set_veh_state(vehicle3, "veh3")
```

2. To support integration groups, I added the integ simulation-object, as detailed in the course notes:

```
S_define
##include "dynamics/dyn_manager/include/dynamics_integration_group.hh"
##include "er7_utils/integration/rk4/include/rk4_integrator_constructor.hh"
class IntegrationSimObject: public Trick::SimObject
{
 public:
  jeod::DynamicsIntegrationGroup group;
  er7_utils::RK4IntegratorConstructor rk4_constructor;
  er7_utils::IntegratorConstructor * rk4_constructor_ptr;
  IntegrationSimObject()
   :
   rk4_constructor_ptr( &rk4_constructor)
   {};
 private:
  IntegrationSimObject (const IntegrationSimObject&);
  IntegrationSimObject & operator = (const IntegrationSimObject&);
};
IntegrationSimObject integ;
```

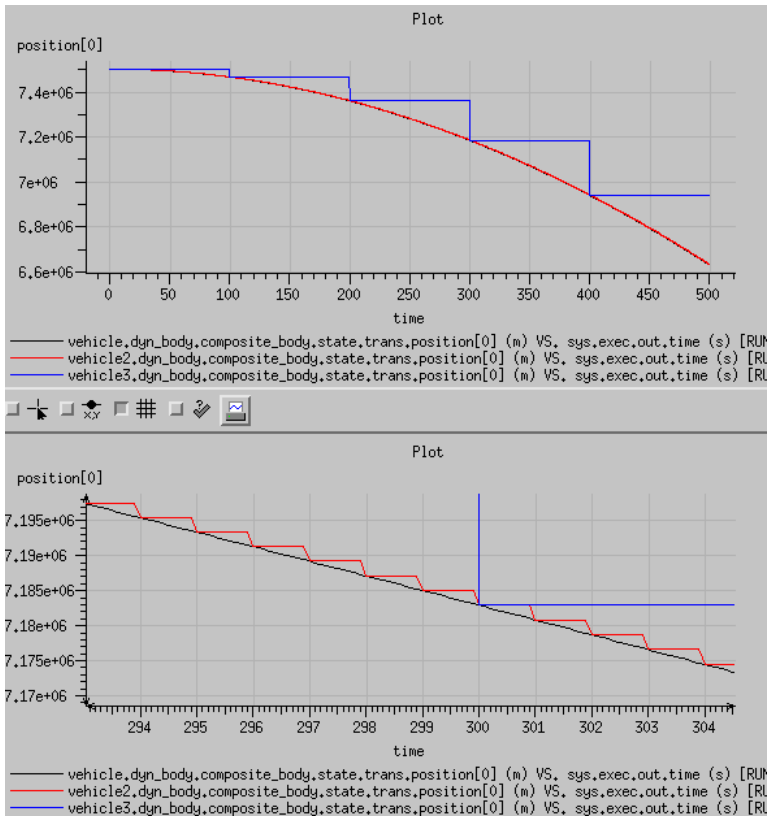
and then created 3 integration groups:

```
JeodIntegLoopSimObject fast(0.005,
                            integ.rk4_constructor_ptr,
                            integ.group,
                            jeod_time.time_manager,
                            dynamics.dyn_manager,
                            env.gravity_manager,
                            &vehicle,
                            NULL);
JeodIntegLoopSimObject med(1.0,
```

```
                              integ.rk4_constructor_ptr,
                              integ.group,
                              jeod_time.time_manager,
                              dynamics.dyn_manager,
                              env.gravity_manager,
                              &vehicle2,
                              NULL);
JeodIntegLoopSimObject slow(100,
                              integ.rk4_constructor_ptr,
                              integ.group,
                              jeod_time.time_manager,
                              dynamics.dyn_manager,
                              env.gravity_manager,
                              &vehicle3,
                              NULL);
```

3. Populated the arguments of the 3 groups with the RK4-integrator defined in the integ sim-object, set the rate accordingly, and passed in a different vehicle to each one.

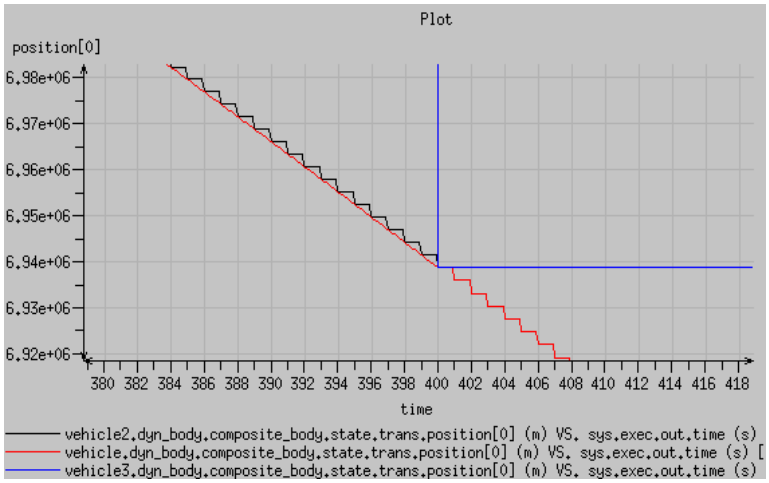
4. Logged data at 0.1s. The following plots show the complete range 0-500s, and a zoomed-in subset around 300s to separate the high-rate and medium-rate plots:



5. Added a trick-event:

```
trick.add_read(400.0, “med.integ_loop.add_sim_object(vehicle)”)
```

The red line (vehicle) is continuous leading up to 400s, and starts to stair-step thereafter, matching the black line (vehicle2).



Exercise 42 Cis-lunar 2-vehicle Simulation and Ephemerides Rate Investigation

1. Largely built off standard S-modules. The exceptions are the addition of the *propagate_lunar_rnp( )* call in MoonSimObject:

```
S_define
31  jeod::De4xxEphemeris  & de4xx;
38    de4xx( environment_in.de405)
40    P_ENV ("derivative") de4xx.propagate_lunar_rnp ( );
```

   and the addition of the three gravity controls to the basic-vehicle simulation-object:

```
S_define
60  jeod::SphericalHarmonicsGravityControls  earth_grav_control;
61  jeod::SphericalHarmonicsGravityControls  moon_grav_control;
62  jeod::SphericalHarmonicsGravityControls  sun_grav_control;
```

2. This is already accomplished in the Modified data files *earth_vehicle.py*, and *moon_vehicle.py*:

```
earth_vehicle.dyn_body.integ_frame_name = "Earth.inertial"
moon_vehicle.dyn_body.integ_frame_name = "Moon.inertial"
```

3. Handle this is two parts – earth-vehicle, and moon-vehicle.

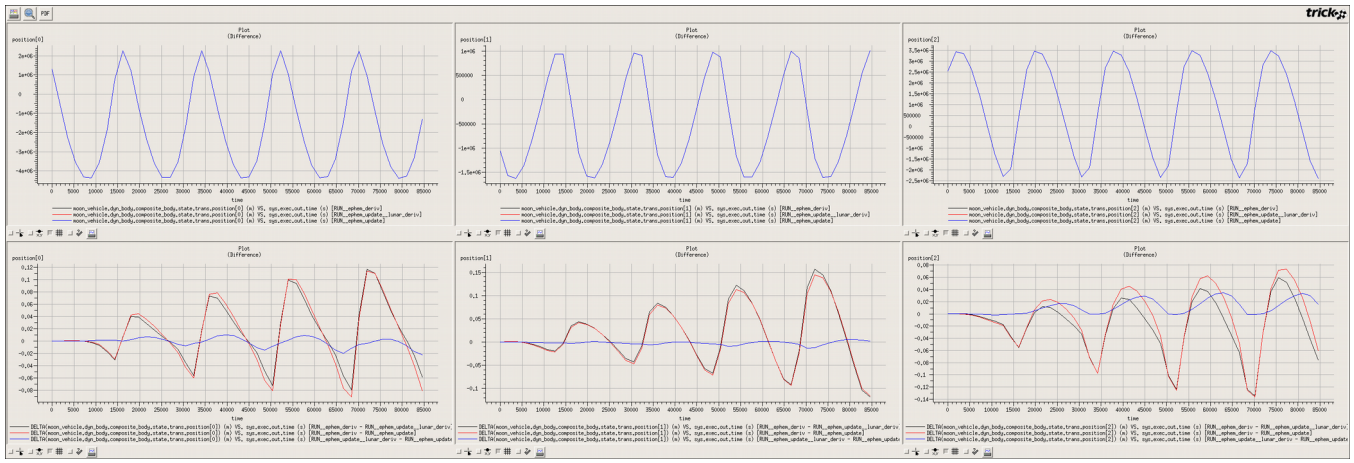
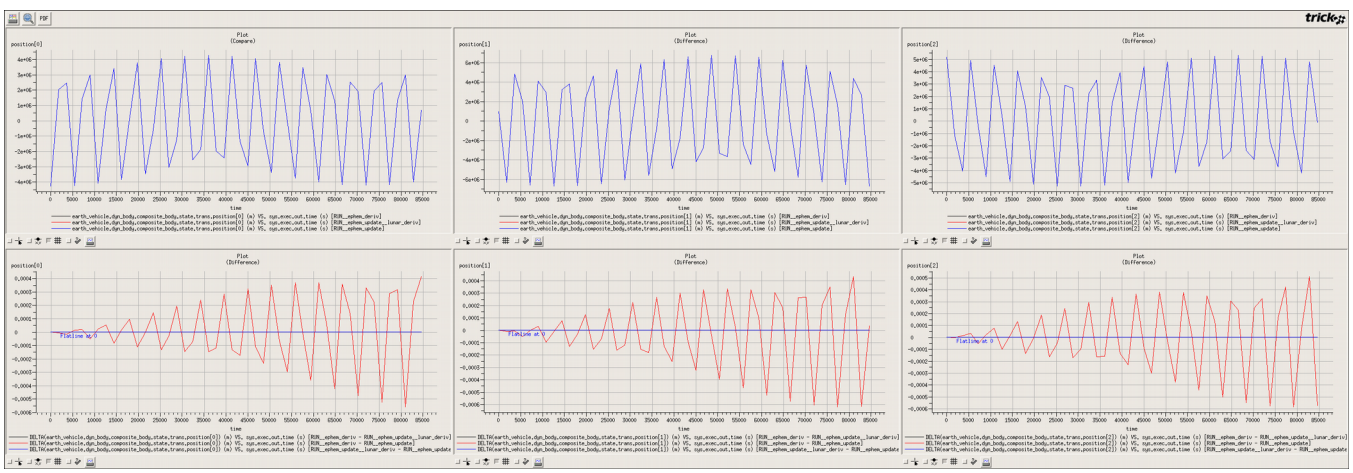
   a) The earth-vehicle is exposed to only a spherical implementation of lunar gravity, so lunar orientation should have no bearing whatsoever. Because the gravity is dominated by earth, knowing the position of the perturbing Moon at the derivative rate is going to be of minor significance.

   b) The moon-vehicle is exposed to a non-spherical implementation of lunar gravity. Lunar orientation should affect the response of the vehicle. Consequently, updating the lunar orientation at the derivative rate should produce a different signature when the rest of the ephemerides are updated only at the dynamics rate. However, when the ephemerides is updated at the derivative rate. Further updating the lunar orientation at the same rate should have no effect whatsoever. Additionally, because earth-gravity is a significant contribution, having the ephemerides update at the derivative rate (which will affect the position of Earth and therefore the response to Earth-gravity) should have a noticeable impact on the overall trajectory.

As expected, running propagate_lunar_rnp( ) is redundant when the ephemerides are running at the derivative rate and

also when lunar gravity is spherical. *moon_vehicle* shows more susceptibility to the update frequency of the ephemerides. Over a 1-day simulation, the peak delta in position (for *moon-vehicle*) between the case with ephemerides running at the dynamics rate and at the derivative rate is approximately 20 cm. Most of that delta is attributable to better knowledge of the planetary positions rather than the lunar orientation; just adding lunar-orienation at the derivative rate has a relatively small effect. In comparison, *earth_vehicle* shows a peak delta in position-delta of less than 1cm on the day, bringing into question the value of running the ephemerides at the derivative rate for an earth-orbiter.

For moon-vehicle:



For earth-vehicle: