JEOD
Training Manual

for

Version 5.0

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

Table of Contents

Chapter A Introduction

A.1 Introduction to JEOD

The JSC Engineering Orbital Dynamics simulation package is used to simulate vehicles in an orbital environment. JEOD provides the ability to model multiple spacecraft trajectories, supporting six-degree-of-freedom propagation of multiple vehicles around multiple planets, and the ability to represent and extract relative states for any simulated vehicle in relation to other vehicles and/or planets.

A.1.1 What JEOD Provides (and does not provide)

JEOD performs the following capabilities:

- Maintains the state of the environment for each simulated vehicle
    - Including gravity, atmosphere, radiation. Electrical and magnetic fields are not included.
    - Allows simultaneous modeling of any number of vehicles, even with distinctly defined environments (e.g. an Earth-orbiting and a lunar-orbiting vehicle)
- Calculates the forces resulting from environment-vehicle interactions
    - Only environmental forces; effectors (including thrusters) are not modeled
- Calculates the forces and impulses resulting from vehicle-vehicle interactions (e.g. collisions).
- Accurately propagates the state of orbital vehicles subject to all forces
    - Forces internally generated within JEOD, and externally applied (e.g. thrusters) may be integrated.
- Modeling is limited to rigid body dynamics
    - Includes discrete relative motion of attached rigid bodies (e.g. articulating elements)
    - The dynamics of flexible bodies is not modeled.
    - Continuum mass motion within a vehicle (e.g. fluid slosh) is not modeled.

A.1.2 Confidence in JEOD

JEOD has been put through extensive inspection, verification, validation, and accreditation processes. Data for regression testing is provided in these verification and validation tests. All models (where such activities make sense) are tested in their respective *verif* directory. Furthermore, integrated simulations are used to compare the whole JEOD package against external data:

- *verif/SIM_dyncomp* provides a test suite comprising systematically more complex scenarios for comparison against other simulation packages.
- *Integrated_Validation* provides cases for comparison against empirical data:

  - Single-vehicle in Earth, Moon, orbits
  - Hyperbolic swing-by orbits past Earth and Mars
  - Two-vehicle simulation using GRACE data

JEOD has been independently certified to Capability Maturity Model Implementation (CMMI) Level 3. Metrics provided include:

- Lines of code
- Extended cyclomatic complexity
- Code coverage

A.2 Information about this Course

A.2.1 Assumed Background

Familiarity with Trick and with the basic Unix/Linux command-line interface is assumed. For this course, we will work in Trick-19. A primer on Trick is provided in Appendix A, and a very brief summary of command-line utilities is provided in Appendix B.

Some knowledge of C++ may be useful, but is not required. A primer on C++ is provided in Appendix C.

Knowledge of how different coordinate frames are defined may prevent confusion later. The document *COORDFRAME.pdf* (released with JEOD at *docs/coordinates/COORDFRAME.pdf*) will help identify acronyms such as LVLH and ECEF, as well as distinguish between body and structure frames.

A.2.2 Where we are Going

By the end of this course, you should have:

- Knowledge of the capabilities of JEOD
- Knowledge of how the different models within JEOD interrelate
- The ability to build an simulation of a reasonably complex scenario
- An understanding of how to manipulate existing simulations to meet your requirements
  - Two of the later exercises include:
    - A vehicle in orbit around Earth, with perturbing gravitational forces from the sun and moon, with effects of aerodynamic drag and radiation pressure included.
    - A two-vehicle simulation with one vehicle in orbit around Earth, and one around Moon.

(See *verif/SIM_dyncomp/S_define* for an example of a complex integrated simulation)

A.2.2.a Key to color-coding

Throughout the course presentation, examples of code are interspersed with descriptions. Examples are color-coded and uniquely bordered for those reading black-and-white copies. At the top of most blocks is the location in the JEOD code-base where the example can be found. Blocks that do not have this location tag are usually a continuation from the previous block

```
This is an example of C++ code or command-line entry
```

```
This is an example of code taken from a simulation-definition (S_define) file
```

```
This is an example of code taken from a simulation input file
```

```
This is an example of code taken from a simulation data file (Modified-data or Log-data)
```

Frequently, the sections of code that are of immediate interest may be separated by multiple lines of code that are not relevant to the discussion at hand. In that case, the unnecessary code is removed and replaced by an ellipsis (...). Also, occasionally you will see that an example is “based on” some actual code. There are two common reasons for not copying the code directly:

1. We pull examples from multiple places in the code base, and sometimes object instance names differ between the different simulations for the same object. In this course, we try to maintain consistency, so variable names may have been changed from what you will see in the actual cited file.

2. The JEOD code-base was developed over a long period of time, and the best practices were refined as the code was developed. Consequently, the code-base does not always conform to best practice. In this course we try to always provide best practice, and where the code-base does not conform, the example provided will be edited so that what you see in this course material does.

A.3 JEOD Overview

A.3.1 Getting Started with JEOD

A.3.1.a How to Get JEOD

JEOD releases are available on GitHub at https://github.com/nasa/jeod.

A.3.1.b Working with JEOD

Bug reports and requests can be submitted as issues on GitHub at https://github.com/nasa/jeod/issues.

Users are welcome to contribute any code that adds to the capability or usability of JEOD with the understanding that this is an open-source project. Contributions can be submitted through pull requests on GitHub at https://github.com/nasa/jeod/pulls.

A.3.2 Learning How to Use JEOD

There are 4 primary sources for information on how to use JEOD:

1. This course (located at *docs/Training*).
    - Provides detailed model-by-model training in the development of integrated simulations.
2. The API document has a plethora of information. It is described in more detail in section html below, and is available from the JEOD release at *html/jeod/index.html*
3. The documentation
    - The top-level document (at *docs/JEOD.pdf*) provides an overview of the package, and is an excellent place to start.
    - All models have their own documentation (at *models/*/*/docs/*.pdf*). All documents contain a User Guide at Chapter 4, which is typically divided into 3 sections:
        - **Analysis**, or **Instructions for Simulation Users** provides an overview primarily for end-users editing the input data
        - **Integration** or **Instructions for Simulation Developers** provides an implementation guide, primarily for simulation developers looking to integrate the model into the simulation.
        - **Extension** or **Instructions for Model Developers** provides a more thorough behind-the-scenes guide, primarily for code developers who need to build on what the JEOD package offers.
    - All models also provide a Reference Manual (at *models/*/*/docs/refman.pdf*). These provide detailed descriptions of the C++ classes included in the model. These documents tend to be large, and are primarily intended as reference manuals for looking up the meaning or context of some specific element of the model. Documents include
        - Class Hierarchy showing inheritance
        - Index to all classes and all files
        - Class-by-class breakdown of all data fields and methods, including UML diagrams.
4. The integrated simulations in the sims top-level directory (*sims*) may be used as models for adaptation to particular situations.

A.3.3 Directory Structure in JEOD

From the top level, the most useful directories are:

A.3.3.a bin

Contains utility scripts.

A.3.3.b docs

Provides the top-level JEOD document, the JEOD coding standards, the materials for this course, and a reference document describing the reference frames used in JEOD.

A.3.3.c html

Contains the JEOD API document. See html/jeod/index.html for the front page. It provides the following information:

- *Main Page* provides links to three dependency diagrams, showing the most significant model-model dependencies for initialization, scheduled, and derivative jobs.
- *Related Pages* provides links to:
  - *README* contains the development history and build instructions for the current release.
  - *Toplevel Documents* provides links to the JEOD top-level document and the Coordinate Frames reference document.
  - *Model Documentation* provides a list of all models, with links to the model-specific documentation.
- *Modules* provides a list of where the various models can be found in the JEOD package. Links provide access to model-specific documentation.
- *Data Structures* provides an alphabetical list of all classes from all models. Links expand the classes to show their contents. Further links provide detailed information on all data fields and methods, including:
  - for data fields, direct links to the defining header file, and links to methods that use that field.
  - For class methods, direct links to the defining source code, links to methods that call said method, as well as methods that are subsequently called from within said method.
- *Files* provides an alphabetical list of all source and header files in the package. Links provide data on dependencies, assumptions, as well as direct access to the code.

Directories provides a hierarchical index to the same information as *Files*.

A.3.3.d lib

Provides a set of pre-packaged S-module files to facilitate rapid development of simulations.This course will make heavy use of these S-modules; it is recommended that simulation developers utiliize these S_modules – either directly or through inheritiance - wherever suitable rather than writing from scratch.

Also provides python scripts to assist with configuration of logging and with checkpoint-restart.

A.3.3.e models

Contains the JEOD code. It is divided into four sections:

- *dynamics* provides the code that ensures JEOD follows fundamental dynamic laws.
- *environment* provides models to simulate environmental conditions.
- *interactions* provides models that dictate how a vehicle will interact with the environment.
- *utils* provides utility functions, such as integrators, reference frames, mathematical techniques, etc.

Within each section, each model defines a subdirectory. Each model subdirectory then typically comprises the following sections (for some models, some sections are not relevant and are omitted):

data					Contains the default data
	include				Contains the headers for the default data classes.
	src				Contains the source code for the default data classes.
include					Contains the headers for the model classes.
src					Contains the source code for the model classes.
verif					Contains the verification data
	src				Additional source code unique to the verification process.
	include				Additional headers unique to the verification process.
	*SIM_**				Directories containing the verification simulations
		S_define			The simulation definition file
		SET_test			Contains the simulation run directories, for testing purposes.
			*RUN_**		Directories containing files that specify the input data for each run of the simulation, and that are generated by the simulation for

					recording simulation output data.
				input.py	The data input file for the simulation run.
		SET_test_val			Contains the same run directories as SET_test, but these directories contain validation data for regression testing.
			*RUN_**		One directory per run, contains the data generated by a nominal run
		Modified_data			Contains all of the Modified data files, possibly arranged in further sub-directories.
		Log_data			Contains the files detailing which variables are to be logged. Note that not simulation runs use all of the files.

A.3.3.f README_jeod

The top-level summary of changes associated with releases.

A.3.3.g sims

Contains miscellaneous simulations of interest (e.g. an abbreviated simulation of an Apollo mission).

A.3.3.h verif

Contains the integrated simulations used for validation against empirical data (in *Integration_Validation*), and *SIM_dyncomp*, a progressive set of simulations of varying complexity used for comparison against other simulation packages.

A.3.4 Error Messages in JEOD

When running a simulation, any anomalies will be flagged by the Message Handler model (see section I.7). There are five primary levels of error:

1. Failure. The simulation will stop
2. Error. Errors almost always invalidate the simulation output, but do not cause the simulation to terminate. If any Error messages are observed, the simulation has problems.
3. Warning. Warnings are sent when a model is being used in a manner in which it was not intended. This does not necessarily mean that the usage is inappropriate, but developers writing simulations that produce warnings should probably have good reason for using the architecture in that way. Reporting of warnings is on by default but may be disabled.

4. Notice. Notices are primarily informational messages – for example a model has performed some unusual or unexpected task that was anticipated during development but for very limited situations. Notices are not usually reported, but may be enabled.

5. Debug. A very low priority message used for debugging purposes and often removed from the code base thereafter. This level is primarily used during code development (as opposed to simulation development) only. Reporting is disabled by default, but may be enabled.

A.4 Trick and JEOD

JEOD 5.0 is compatible with Trick-19. In this section, we will cover how to use JEOD in a Trick environment.

Trick uses certain path specifications to find models that are instantiated in the S_define file (or S_module files). These can be set as environment variables, or in the file S_overrides.mk at the same location as the S_define.

- **TRICK_CFLAGS,** and **TRICK_CXXFLAGS** provide paths to be searched for C and C++ code. Typically, the models directory would be added to these settings:

```
S_overrides.mk:
TRICK_CFLAGS += -I${JEOD_HOME}/models
TRICK_CXXFLAGS += -I${JEOD_HOME}/models
```

  Then the model header files can be included relative to that path:

```
S_define:
##include "dynamics/dyn_body/include/simple_6dof_dyn_body.hh"
```

- **TRICK_S_FLAGS** provide paths to be searched for *S-modules*. Typically, the *lib/jeod or lib/jeod/JEOD_S_modules* directory would be added to this setting. Then the pre-packaged *S-module* files can be included directly:

```
S_overrides.mk:
TRICK_SFLAGS += -I${JEOD_HOME}/lib/jeod/JEOD_S_modules
```

```
S_define:
// JEOD S-modules
#include "dynamics.sm"
```

  OR

```
S_overrides.mk:
TRICK_SFLAGS += -I${JEOD_HOME}/lib/jeod
```

```
S_define:
#include “JEOD_S_modules/dynamics.sm”
```

A.4.1 Initializing JEOD models

A.4.1.a Sequencing of Calls

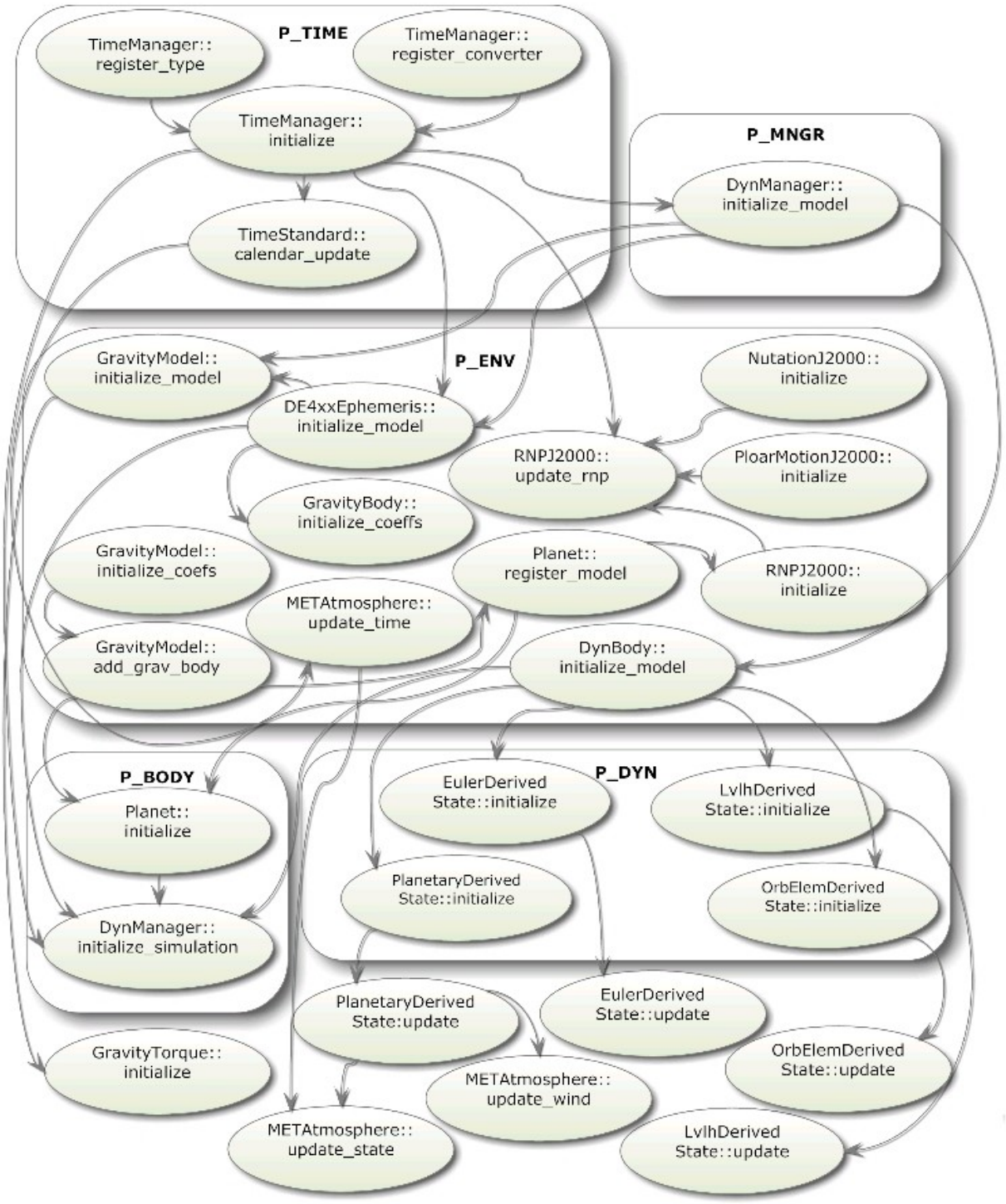
To run a JEOD simulation, the JEOD models must be properly initialized. Some initialization methods rely on the completion of others. As a rough guide, the recommended order is:

1. Time
2. Dynamics Manager
3. Environment-related models
4. Vehicle
5. Relative frames and states

A somewhat detailed (but not exhaustive) graphical illustration of the recommended call sequence is available in the API document (at *html/jeod/index.html*) and is included here for additional reference. This diagram shows the specific function calls for the integrated simulation SIM_dyncomp (*verif/SIM_dyncomp*), which are identified in an S_define as *“initialization”* class job calls.

```
verif/SIM_dyncomp/S_define
110       P_TIME  ("initialization") manager.register_type(
111           tai );
112       P_TIME  ("initialization") manager.register_converter(
113           conv_dyn_tai );
```

In general, a good starting place for constructing a new S_define is to follow practice from the default simulation modules and the worked examples provided in this course. . Where those are insufficient, the other integrated (verification) simulations, and the model-specific verification simulations are also a good source. Be aware, however, that the JEOD code-base was built over an extended period of time while development was ongoing in both JEOD and Trick. Consequently, the methods used – particularly in the model verification simulations – are not always optimal methods. Where conflicts arise, the methods and assignments used in the default simulation modules and this course should be assumed to be most correct.



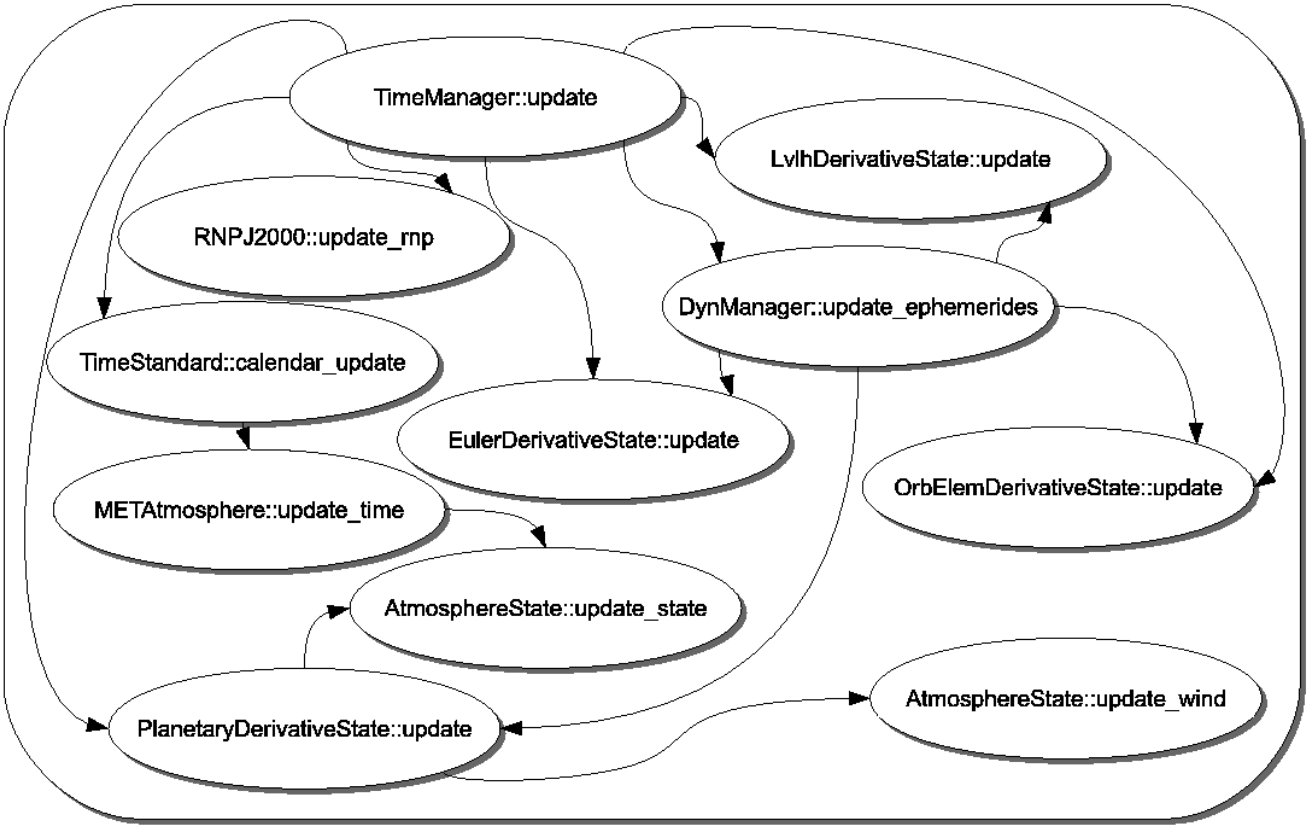
Trick Initialization Job Dependencies

A.4.2 Function Calls Associated with Updating JEOD Models

Since much of JEOD is associated with modeling the environment in which a vehicle moves, most routine update function calls are identified in the S_define with the keyword *"environment"*

```
lib/jeod/JEOD_S_modules/Base/jeod_time_base.sm
 26       (DYNAMICS, "environment") time_manager.update ( exec_get_sim_time());
```

A sample of the dependencies is shown, again from *SIM_dyncomp*:



Trick Scheduled Job Dependencies

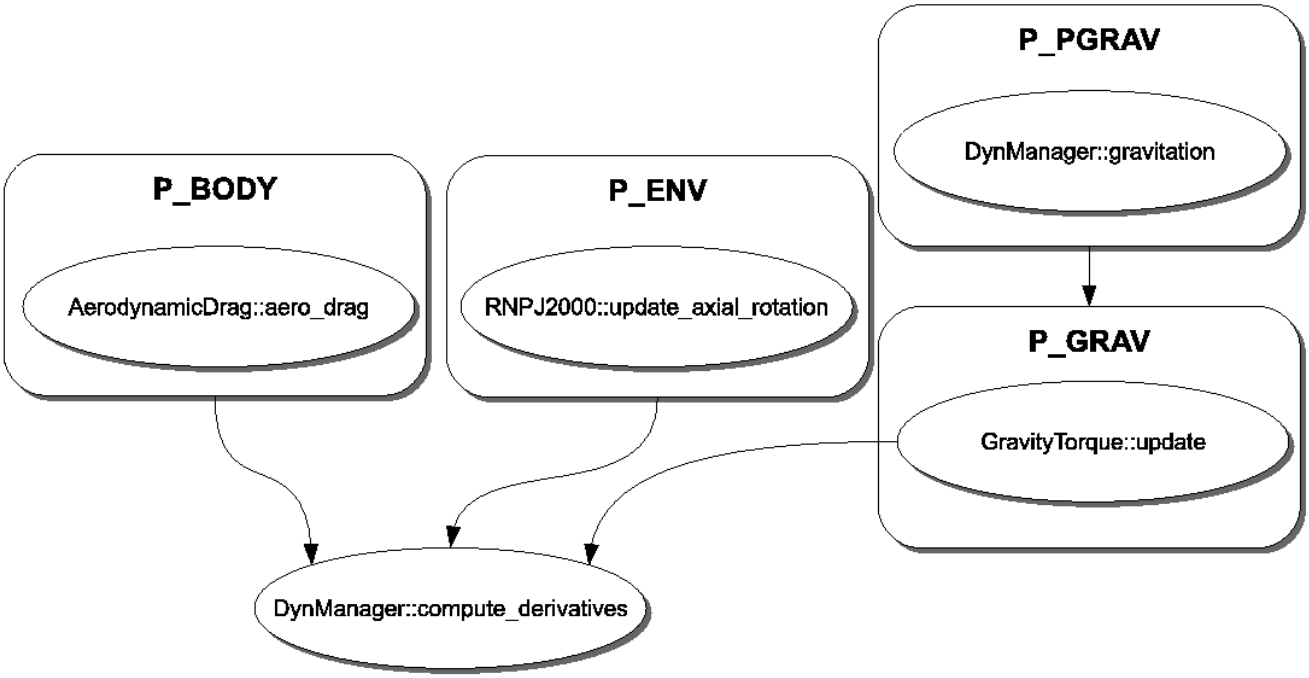
A.4.3 Function Calls Associated with Integrating Vehicle Dynamics

Environmental conditions that change rapidly may need to be updated within an integration step (when the integration algorithm uses multiple stages within one step, e.g. RK4). These calls are identified with the *"derivative"* calling priority. For JEOD applications, there is only one (1) integration call; in most cases it is made to the Dynamics Manager *integrate* method. The Dynamics Manager knows of all vehicles in the simulation and can integrate everything from this one call.

```
lib/jeod/JEOD_S_modules/Base/dynamics.sm
52      P_GRAV ("derivative") dyn_manager.gravitation ( );
53      ("derivative") dyn_manager.compute_derivatives ( );
54
55      //
56      // Integration jobs
57      //
58      ("integration", &dyn_manager.sim_integrator) trick_ret =
59      dyn_manager.integrate ( exec_get_sim_time(),
60         time_manager);
```

In simulations involving multiple integrators, the call is slightly different. More on that when we look at advanced integration capabilities (section I.4).

A sample of the dependencies for derivative-class jobs is shown here, again from *SIM_dyncomp*:



Trick Derivative Job Dependencies

A.4.4 Structuring a Simulation into Simulation Objects

When defining a simulation, we break the scenario into multiple easily-managed objects. It is desirable – at least in principle – to create these objects such that they represent self-contained clusters of data.

An integrated simulation may, for example, define objects that simulate:

- Time
- The dynamics of all of the vehicles
- The environment
- Each planet
- Each vehicle
- The interaction between the vehicles and the environment.

The reality is that completely separating out the operation of one class from all other classes is just not possible. There is always going to be some inter-dependency (or we would have a series of simulations instead of an integrated simulation!)

The standard method used by JEOD for accessing data found in one object from another object is to pass a reference to the one into the other at construction time. The consequence is that the dependency can only go one way (one class must be constructed in order to pass its data into the other).

An example that is repeated in all simulations is the dependency of the dynamics on time. First, the class associated with the Time object is instantiated, it has no dependency so usually appears first in the S_define.

```
lib/jeod/JEOD_S_modules/Base/jeod_time_base.sm
 15 class JeodTimeBaseSimObject: public Trick::SimObject {
...
 27 };
 lib/jeod/JEOD_S_modules/time_dyntime_only.sm
 2 JeodTimeSimObject jeod_time;
```

In this case, the instance of the *JeodTimeSimObject* is called *jeod_time*. The next class instantiated is usually the Dynamics Manager class, because it has a dependency on time only. Because it has that dependency, we first create a **private** reference to a *TimeManager* object:

```
lib/jeod/JEOD_S_modules/Base/dynamics.sm
...
 21 class DynamicsSimObject: public Trick::SimObject {
...
 23 protected:
 24    // Reference to the external time manager.
 25    jeod::TimeManager & time_manager;
```

then pass in an argument of that type at construction

```
  lib/jeod/JEOD_S_modules/dynamics.sm
  2 DynamicsSimObject dynamics (jeod_time.time_manager);
```

NOTES:

1. some of the older verification simulations access external data with the use of pointers instead of references. These simulations are old, and this practice was phased out when the ability to pass references became readily available. Using pointers will work, but it is more cumbersome to set up, and not as secure.

2. The reference must be private, this will not work with public references.

Because references can only be populated at construction, we must set the internal *time_manager* value to the argument in the constructor. First, the constructor is defined to take a *TimeManager* class argument, temporarily called *time_manager_in:*

```
lib/jeod/JEOD_S_modules/dynamics.sm
35    // Constructor
36    DynamicsSimObject (
37       TimeManager & time_manager_in)
```

the temporary *time_manager_in* is immediately assigned to the reference value *time_manager* in the initialization list:

```
38    :
39       time_manager (time_manager_in)
40    {
```

then the constructor continues.

By this method, the object called *dynamics* has an element called *time_manager* which is effectively equivalent to the argument passed in. Thus, the entire contents of *jeod_time.time_manager* can be accessed from *dynamics.time_manager*.

A.4.5 Default S_modules

To assist with creating S_define files, the JEOD package includes a small set of functional examples of S_define objects. These are all located at *lib/jeod/JEOD_S_modules/*. We have already seen elements of some of these in the examples presented thus far. The modules are provided in two flavors: the instantiated modules and the base modules. More on that later.

Using default S-modules:

The most convenient way to include these in an S_define is to add the directory to the TRICK_SFLAGS setting, as described in section Trick and JEOD on page 21.

Alternatively, users could provide a symbolic link from the simulation directory to the JEOD S_modules directory, then access the files through the link. This method is useful for simple scenarios for which an S_overrides.mk file is not necessary, but for most projects an S_overrides.mk file is already necessary to define the path to the JEOD models.

```
>> ln -s $JEOD_HOME/lib/jeod/JEOD_S_modules JEOD_S_modules
```

```
S_define
#include "JEOD_S_modules/jeod_sys.sm"
```

NOTE – if the training module is added to the JEOD workspace, thereby creating a training directory at the same level as bin, docs, html, etc, then the command-line for creating the link from each exercise area is (e.g. for SIM_01):

```
cd training/Exercises/SIM_01
ln -s  ../../../lib/jeod/JEOD_S_modules JEOD_S_modules
```

Instantiated Modules and Base Modules:

The instantiated modules are those found directly at *JEOD_S_modules*, e.g. *earth_basic.sm*, *environment.sm*, *dynamics.sm.* Each of these modules *provides* an instance of the module, so the inclusion of one of these files into the S_define is all-encompassing. For example,

```
#include “dynamics.sm”
```

defines and instantiates the dynamics sim-object. Many of these files are very simple front-ends to the base simulation-objects, which are found in the *Base* directory. For example, dynamics.sm is just 2 lines:

```
#include "Base/dynamics.sm"
DynamicsSimObject dynamics (jeod_time.time_manager);
```

Unlike the instantiable modules, the base modules **do not** include any instantiations of their content; they simply define the simulation-object classes, including all of the necessary header file inclusions, data declarations, and initializations that are often the most time-consuming element of setting up basic simulations. These can be used as base-classes for custom simulation-objects. To inherit one of these classes as the basis for a user-defined class, follow these steps (an example follows):

1. include the necessary headers for the sim-object elements that you want to *add* to the base-module.
2. #include the base module file.
3. Start the definition of the new class. For the parent class, identify the base module class instead of the more usual Trick::SimObject class.
4. Add only the content that is not already a part of the base-module
5. The constructor should take as arguments all external data/references that are needed for the base-module *and* any that are needed for the new additions. In the constructor's initializer list, remember to call the base-module's constructor with appropriate arguments.

There are cases in the Base directory where more complex base-modules are built on more primitive base-modules. As an example, consider what happens when the earth_GGM02C_MET_RNP.sm file is included into a S_define:

```
#include “ earth_GGM02C_MET_RNP.sm”
```

If we look at this file, we see it is a simple 2-line file: the inclusion of a base module and its instantiation:

```
#include "Base/earth_GGM02C_MET.sm"
Earth_GGM02C_MET_SimObject earth( dynamics.dyn_manager,
                                  env.gravity_manager,
                                  jeod_time.time_utc,
                                  jeod_time.time_tt,
                                  jeod_time.time_ut1,
                                  jeod_time.time_gmst);
```

The base-module *earth_GGM02C_MET.sm* starts with the inclusion of the necessary headers for the atmosphere, then these two lines:

```
#include "earth_GGM02C.sm"
class Earth_GGM02C_MET_SimObject : public Earth_GGM02C_SimObject
```

For those users not familiar with C++ inheritance, this means that the *Earth_GGM02C_MET_SimObject* has all the capabilities of the *Earth_GGM02C_SimObject*, which is defined in file *Base/earth_GGM02C.sm*. This file starts by including the header files necessary for the RNP model, and earth data, then these two lines:

```
#include "Base/planet_generic.sm"
class Earth_GGM02C_SimObject: public PlanetGenericSimObject
```

Thus, our new class, *Earth_GGM02C_MET_SimObject*, has all the capabilities of the *Earth_GGM02C_SimObject*, which in turn has all the capabilities of the *PlanetGenericSimObject*, which is defined in *Base/planet_generic.sm.*

The *PlanetGenericSimObject* inherits from *Trick::SimObject*, so the sim-object inheritance hierarchy ends there. But going in the other direction, there is no requirement that *Earth_GGM02C_MET_SimObject* be terminal. Users could, define their own EarthSimObject that inherits all the content of *Earth_GGM02C_MET_SimObject* and adds, for example, ground stations, or albedo models, or other atmosphere models.

The Instantiated S-modules:

jeod_sys

The most important is *jeod_sys.sm.* This module is absolutely required for all simulations that include any JEOD elements, even simple unit-test simulations. It should not be modified.

integration

The *integ_loop.sm* module is used for simulations requiring advanced integration concepts (we will cover this in Section I.4). It should not be modified.

default_priority_settings

If any of the other simulation modules (with the exception of *jeod_sys.sm* and *integ_loop.sm*) are used, this module must be included. We have already seen that some function calls are dependent on others already having been made; Trick provides a prioritization capability that allows the user to specify when methods are to be called. In the following simulation modules, we use certain abbreviations (e.g. *P_TIME*) that are defined in this module.

Note that users developing large-scale simulations may need to provide additional priority settings. Managing these priorities is greatly simplified if all priorities are found in one location. There are two simple methods for managing this:

1. Copy *default_priority_settings.sm* into a project-specific location and add to that project-specific file as necessary. In the simulation, include only the project-specific file. Note that it is vey important in this case that the JEOD_specified values do not get rearranged.

2. Copy *default_priority_settings.sm* into a project-specific location, comment out the JEOD-settings and add to that project-specific file as necessary. In the simulation, include both the JEOD *default_priority_settings.sm* file and the project-specific file.

Another necessary setting that is not included in default_priority_settings is that of the *DYNAMICS* rate. Several of the default S-modules use the *DYNAMICS* keyword to schedule their executive jobs, so this value must be defined to the desired value in the S_define.

```
S_define:
#define DYNAMICS 1.0
```

NOTE – if it is desired to run these jobs at different rates, (e.g. update time at 1.0 and atmosphere at 5.0), the S_modules must be copied and edited accordingly.

dynamics

dynamics.sm, *dynamics_init_only.sm*, and *dynamics_multi_group.sm* provide implementations of the dynamics manager simulation object.

- ***dynamics.sm*** is the most frequently used sim-module. It includes the integration-class job-call to perform the integration in the case that there is only one integration group (default operation, for more information on integration groups, see section I.4).

- ***dynamics_init_only.sm*** only includes the initialization process. It has no integration calls nor environment update calls, so is suitable for unit-tests.

- ***dynamics_multi_group.sm*** also omits the integration-class job-call, but does include ephemerides and events calls. It is intended to be used to support simulations involving multiple-integration-groups (see section I.4).

environment

environment.sm and *environment_sans_de405.sm* provide, respectively, an environment framework that includes the ephemerides model, and one that does not (the ephemerides model dictates the positions and orientations of the planets as a function of time). A simulation with no planets, or one in which the planetary positions are either irrelevant or integrated rather than generated from ephemeris data should use *environment_sans_de405.sm.* Most simulations would use *environment.sm.*

Note – the name *environment_sans_de405.sm* is somewhat misleading, a more correct name would be *environment_sans_de.sm.* JEOD provides DE405 and DE421 ephemerides models; this simulation-module includes neither; whereas *environment.sm* allows the user to choose either.

planetary

There are three modules for simulating Earth; simulation-developers should select only one of these.

- *earth_basic.sm* creates a simulated Earth with a spherical gravitational field.

- *earth_GGM02C.sm* replaces the spherical gravitational field with one from the GGM02C (non-spherical) model.
- *earth_GGM02C_MET_RNP.sm* is the most advanced; this Earth has a GGM02C gravitational field, a MET model atmosphere, and an RNP (rotation-nutation-precession) model for Earth orientation.

For Moon:

- *moon_basic.sm* creates a simulated Moon with spherical gravity.
- *moon_lp150Q.sm* creates a simulated Moon with the LP150Q (Lunar Prospector 150x150) non-spherical gravity model.

For Sun:

- *sun_basic.sm* creates a simulated Sun with spherical gravity.

time

The time model, while flexible and versatile, is consequentially cumbersome to set up. There are several time modules provided:

- *time_dyntime_only.sm* includes no clocks for absolute timekeeping. It counts only from 0 at simulation start.
- *time_TAI_UTC_UT1.sm* adds the TAI, UTC, and UT1 clocks (the three most widely used clocks). All three clocks are expressed in a decimal representation only.
- *time_TAI_UTC_UT1_calendar.sm* adds the calendar representations to UT1 and UTC.
- *time_TAI_UTC_UT1_TT.sm* provides the TAI, UTC, UT1, and TT clocks. TT is used in the ephemerides model, so must be included if that model is used.
- *time_TAI_UTC_UT1_TT_calendar.sm* provides the TAI and TT clocks as decimal representations only, and the UT1 and UTC in both decimal and calendar formats.
- *time_TAI_UTC_UT1_TT_GMST.sm* provides the TAI, UTC, UT1, TT, and GMST clocks. GMST (sidereal time) is used in the RNP model, so must be included if that model is used.
- *jeod_time.sm* is an all-encompassing module that relies on the setting of certain environment variables to access the different clocks. The TAI clock is included by default, then the other clocks are included if their appropriate environment variable is defined. All of the *TIME_MODEL_** flags indicate that the clock should be included in decimal form, and updated at the *DYNAMICS* rate. The value of the environment variable is usually not important. The variables tested are:
  - *TIME_MODEL_UTC*
  - *TIME_MODEL_UT1*
  - *TIME_MODEL_TDB*

- *TIME_MODEL_TT*
- *TIME_MODEL_GMST*
- *TIME_MODEL_GPS*
- *TIME_CALENDAR_UPDATE_INTERVAL* This is the only variable for which the value is significant; it is equal to the time interval between calls to the calendar update for the UT1 and UTC clocks.

vehicle

- *vehicle_basic.sm* provides an instance of a basic vehicle – an entity with mass, state, and ability to identify the resolution on the gravity field.
- *vehicle_atmosphere.sm* provides a more complex vehicle for cases where aerodynamic drag is important; it includes an atmospheric state.

The Base S-modules

The dynamics, earth, environment, moon, sun, and vehicle sim-objects are the back-end to the respective instantiated sim-objects already discussed. While naming conventions are not identical, they are similar.

The exceptions are time and planet-generic. *jeod_time_base.sm* is the back-end for the front-end *time_dyn_time_only.sm*. *jeod_time.sm* is the back-end for all of the other front-end time modules. In addition, there are other time modules that have fixed inclusions of certain clocks; these may be desirable for specific applications.

Planet-generic is the only provided S-module that is not instantiable by itself. It requires planetary data and gravity-field data to be constructed; these are provided by all of the earth, moon, and sun S-modules.

Exercise 1. Familiarity with the S_define

A.5 C++ and JEOD

This section covers the recommended techniques for navigating the C++ code-base in the JEOD package. Probably the most frequent need for this information is to identify the purpose of a particular variable. For most users, this section can be skipped and studied later as needed.

A basic knowledge of C++ is required for an understanding of this section. See Appendix C for an overview of C++, and try Exercise 2 to confirm your understanding of abstract classes.

Exercise 2. Investigating Abstract Classes

A.5.1 Tracing Inheritance in JEOD

The most versatile tool for investigating the data hieraarchy is the API document (html/jeod/index/html)

A.5.1.a Tracing Within a Model

The process is best shown with an example.

Suppose we want to find the meaning of the variable *jeod_time.manager.dyn_time.seconds.*

First, we look at the S_define for the top-level item, *jeod_time.* We find that within this S_define object, there is an instance of *TimeManager*, called *manager.* The variable must be contained within that class. In the API document, we select *Data Structures*, and from the list, choose *TimeManager*.

TimeIntegrator	The base class for classes that propagate time
TimeManager	To manage the various time representations and the converters between them throughout the simulation
TimeManagerInit	To initialize the Time Manager

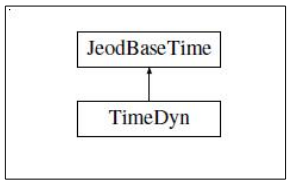
Now we are looking for the instance called *dyn_time*, which is found to be an instance of the *TimeDyn* class. Following the link, we go to *TimeDyn*, and look for *seconds*.

Data Fields

double	**simtime** simulation time (sys.exec.out.time).
TimeDyn	**dyn_time** The instance of **TimeDyn**, the dynamic time that is used as the integration time.
int	**num_types**

Sometimes, such as now, the variable is not going to be there. That does not mean it is not an element of that class, just that it is not defined in that particular header file. That means that this particular class must have inherited the variable from its parent. In some versions, the inheritance graph is easy to find, in others it does not show up.

Where it does, it looks like:



If this is not visible, there should be a link to the header file:

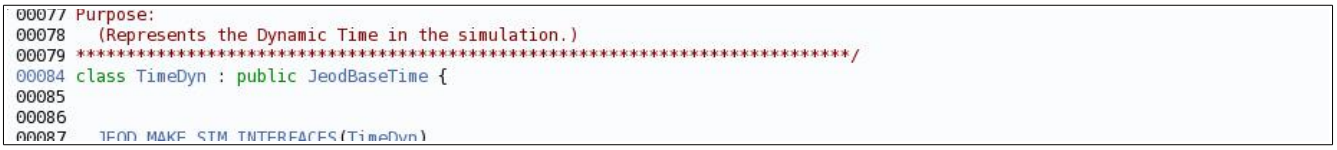

TimeDyn Class Reference
Time

Represents the Dynamic Time in the simulation. More...

#include <time_dyn.hh>

Inheritance diagram for TimeDyn:

which brings up the class, with its immediate parent as a clickable link:


```
00077 Purpose:
00078   (Represents the Dynamic Time in the simulation.)
00079 *******************************************************************************/
00084 class TimeDyn : public JeodBaseTime {
00085
00086
00087   JEOD_MAKE_SIM_INTERFACES(TimeDyn)
```

In either case, we see that *TimeDyn* inherits from *JeodBaseTime*. Clicking on the parent (*JeodBaseTime*) takes us to its class page, where we find *seconds*:


determines which converter function (a_to_b (+1) or b_to_a (-1)) to use.

double **seconds**
elapsed time from epoch
Units: s

char * **name**

Hence, we identify *seconds* as being the elapsed time from epoch. If the original reference came from within JEOD, then we can verify that we have the correct variable by clicking on the variable itself and reviewing the list of places where it is used.


double JeodBaseTime::seconds

elapsed time from epoch
Units: s

Definition at line **124** of file **time.hh**.

Referenced by **TimeUDE::clock_update()**, **TimeConverter_TAI_TT::convert_a_to_b()**, **TimeConverter_TAI_GPS::convert_a_to_b()**, **TimeConverter_Dyn_TAI::convert_a_to_b()**, **TimeConverter_TAI_TDB::convert_a_to_b()**, **TimeConverter_Dyn_UDE::convert_a_to_b()**, **TimeConverter_STD_UDE::convert_a_to_b()**, **TimeConverter_TAI_TT::convert_b_to_a()**, **TimeConverter_TAI_GPS::convert_b_to_a()**, **TimeConverter_STD_UDE::convert_b_to_a()**, **TimeStandard::convert_from_calendar()**, **De4xxEphemeris::ephem_update()**, **PropagatedPlanet::ephem_update()**, **TimeConverter_Dyn_TAI::initialize()**, **TimeConverter_Dyn_UDE::initialize()**, **TimeConverter_STD_UDE::initialize()**, **TimeUDE::initialize_from_parent()**, **TimeStandard::initialize_from_parent()**, **TimeDyn::initialize_initializer_time()**, **TimeUDE::initialize_initializer_time()**, **TimeStandard::initialize_initializer_time()**, **JeodBaseTime()**, **De4xxEphemeris::propagate_lunar_rnp()**, **TimeConverter_Dyn_UDE::reset_a_to_b_offset()**, **TimeConverter_STD_UDE::reset_a_to_b_offset()**, **TimeStandard::seconds_of_year()**, **TimeUDE::set_initial_times()**, **TimeUDE::set_time_by_clock()**, **set_time_by_days()**, **set_time_by_seconds()**, **TimeGPS::set_time_by_trunc_julian()**, **TimeStandard::set_time_by_trunc_julian()**, **TimeDyn::update()**, **RNPMars::update_axial_rotation()**, **RNPJ2000::update_axial_rotation()**, **DynBody::update_integrated_state()**, **TimeDyn::update_offset()**, **RNPMars::update_rnp()**, and **RNPJ2000::update_rnp()**.

A.5.1.b Tracing Across Models

When a class inherits from another model, or contains an instance of a class from another model, the trail can be a little longer, but it follows the same methodology.

Example:

A vehicle is defined in the S_define as being a *DynBody*. What does its *mass.composite_properties.position* represent?

Again, we start in the API at Data Structures, and select *DynBody.*

DynBody does not have a data instance called *composite_properties*, but it has a *MassBody* (*mass)* that does. It is an instance of *MassProperties.*

MassProperties inherits from *MassBasicPoint*, which inherits from *MassPointState*, which contains a double called *position*, described as being the "Mass point location with respect to the origin of some parent frame and expressed in the parent frame's coordinates":

Exercise 3. Tracing Inheritance

Chapter B Basic Building Blocks - Introduction

The chapter covers the basic building blocks for a simulation, presented in the order in which they are typically found in a *S_define* file. By the end of this chapter, you should be able to build a detailed simulation with minimal capabilities.

B.1 Standard Simulation-modules

B.1.1 System Simulation-modules

For all Trick simulations, the *trick_sys.sm* file must be included in the S_define.

For all JEOD simulations, the *jeod_sys.sm* file must be included in the S_define.

```
verif/SIM_dyncomp/S_define
 42 #include "sim_objects/default_trick_sys.sm"
 43 #include "JEOD_S_modules/jeod_sys.sm"
```

We will not cover the contents of either of these in this course.

B.1.2 JEOD Pre-packaged Simulation-modules

The JEOD release includes a collection of pre-packaged simulation-modules at *lib/jeod/JEOD_S_modules.* These files provide self-contained simulation-modules, or they may be used as templates for specific tailoring to meet project needs. A description of these files is provided in section A.4.5.

B.2 Time

Document reference: *models/environment/time/docs/time.pdf*

Perhaps the single most important realization in the JEOD time models is that JEOD maintains its own timekeeping, and it is the JEOD time that is used for integration, not the time maintained by the simulation engine. The Time model must be included in the S_define in order for a JEOD simulation to function.

B.2.1 Concepts of Time

B.2.1.a Dynamic Time

The time model introduces a new concept, that of **dynamic time** (*dyn_time*) which is used for the purpose of dynamics integration. All clocks flow down from the advance of dynamic time. Dynamic time is simulation-specific. It always starts at the beginning of a simulation with the value 0.0, and may advance forwards (default) or backwards (if needed).

The simulation engine (e.g. Trick) will also likely have an internal clock/counter that is used for scheduling function calls and events. We refer to this time as the *simulation-time.* Dynamic time advances in some pre-

determined way with respect to simulation-time, and in most applications they can be considered equivalent. However, JEOD sees the simulation-time as a dimensionless counter, and ascribes a time-dimensioned value (e.g. 1 second, 2 hours, etc.) to each integer 'tick' of simulation-time.

B.2.1.b Standard Clocks

Standard clocks are those that are universally-defined so that a particular value of a clock has a definite specific meaning subject to an understood epoch. When standard clocks are provided, dynamic time must be anchored to some standard clock so that the advance of dynamic time can be propagated to all standard clocks. JEOD provides the following standard clocks:

- TAI – International Atomic Time (Temps Atomique International). This one ticks with dynamic time, so effectively defines the integration clock.
- UTC – Coordinated Universal Time.
- UT1 – principal form of Universal Time.
- TT – Terrestrial Time.
- GPS – Global Positioning System time.
- GMST – Greenwich Mean Sidereal Time.
- TDB – Barycentric Dynamic Time.

JEOD is easily extensible to provide others as needed.

B.2.1.c User-Defined-Epoch Clocks

In contrast to a standard clock, a user-defined clock often has application only in the current simulation. A commonly used example of such is Mission Elapsed Time. The user must specify the epoch of these clocks relative to either dynamic time or some standard clock.

There is a special class for Mission Elapsed Time which allows for the clock to be paused at will, while regular User-Defined-Epoch clocks must continue running just as standard clocks do.

B.2.2 S_define setup

While the Time model is versatile and flexible, that comes with the expense of being tricky to generate the S_define file. A number of simulation modules are provided; users may find it easiest to experiment with those before trying to generate their own S_define. The main difficulty comes in remembering to add all of the calls as the time S_module can get large. Apart from the length, the pattern behind the structure of the S_define is actually very simple.

B.2.2.a Time Declarations

There are four steps to the declarations:

1. Declare the Time Manager and its Initializer

```
lib/jeod/JEOD_S_modules/Base/jeod_time.sm
 70    // The time manager and its initializer
 71    jeod::TimeManager time_manager;
 72    jeod::TimeManagerInit time_manager_init;
```

2. Declare any desired clocks (dynamic time is free with the Time Manager)
    - Because standard clocks are universally-defined, there is no need to have more than one of any type; multiple instances of standard clocks is not permitted.

```
lib/jeod/JEOD_S_modules/Base/jeod_time.sm
 75    jeod::TimeTAI time_tai;
...
 80    jeod::TimeUTC time_utc;
...
 88    jeod::TimeUT1 time_ut1;
```

    - Conversely, there is no restriction on the number of user-defined clocks.

```
models/environment/time/verif/SIM_5_all_inclusive/S_define
 71     jeod::TimeMET  metveh1;
 72     jeod::TimeMET  metveh2;
```

3. Declare all necessary converters
    - Every clock needs an instance of a converter for generating its value. The converters for the standard clocks are defined one class at a time; there should be only one instance of each standard-to-standard converter.

```
lib/jeod/JEOD_S_modules/Base/jeod_time.sm
 76    jeod::TimeConverter_Dyn_TAI time_converter_dyn_tai;
...
 81    jeod::TimeConverter_TAI_UTC time_converter_tai_utc;
...
 89    jeod::TimeConverter_TAI_UT1 time_converter_tai_ut1;
```

    - The converters for the user-defined clocks are more generic, there may be multiple instances of this class type:

```
models/environment/time/verif/SIM_5_all_inclusive/S_define
 80     jeod::TimeConverter_STD_UDE  tai_metveh1;
 81     jeod::TimeConverter_STD_UDE  tai_metveh2;
```

4. Declare any default data (data tables) that are used in the conversions. This is only valid for generating UTC (needs leap-second tables) and UT1 (needs empirical TAI-UT1 drift data, UT1 is not an analytically-calculable quantity).

```
lib/jeod/JEOD_S_modules/Base/jeod_time.sm
 82    jeod::TimeConverter_TAI_UTC_tai_to_utc_default_data
 83       time_converter_tai_utc_default_data;
...
 90    jeod::TimeConverter_TAI_UT1_tai_to_ut1_default_data
 91       time_converter_tai_ut1_default_data;
```

Important note – the order in which the clocks is declared is not important. The declaration of the clocks can be interspersed with other declarations (as was the case in this example).

B.2.2.b Time Initialization

Time is typically the first model initialized because it has no dependencies but numerous dependents. Again, as with the declaration, the file can be long but the structure is simple.

1. Register each of the declared clocks with the Time Manager. This is a TimeManager call with the clock passed as the argument
    - The standard clocks are pre-configured and use the *register_type* function call.

```
lib/jeod/JEOD_S_modules/Base/jeod_time.sm
122       P_TIME  ("initialization") time_manager.register_type ( time_tai);
...
129       P_TIME  ("initialization") time_manager.register_type ( time_utc);
...
137       P_TIME  ("initialization") time_manager.register_type ( time_ut1);
```

    - The user-defined clocks must be named, and use the *register_multi_type* function call with an additional argument to associate the clock with a name.

```
models/environment/time/verif/SIM_5_all_inclusive/S_define
117       P_TIME  ("initialization") manager.register_multi_type(
118           metveh1,
119           "met_veh1");
...
124       P_TIME  ("initialization") manager.register_multi_type(
125           metveh2,
126           "met_veh2");
```

2. Register each of the converters with the Time Manager in a similar way.
    - The converters for the standard clocks are pre-configured; use the *register_converter* function call.

```
lib/jeod/JEOD_S_modules/Base/jeod_time.sm
123       P_TIME  ("initialization") time_manager.register_converter (
124                                                   time_converter_dyn_tai);
...
130       P_TIME  ("initialization") time_manager.register_converter (
131                                                   time_converter_tai_utc);
...
138       P_TIME  ("initialization") time_manager.register_converter (
139                                                   time_converter_tai_ut1);
```

- The converters for the user-defined clocks must specify which clocks they are converting from and to. Two additional arguments - the names of the clocks – are passed with the *register_generic_converter* function call so that the *TimeManager* can connect these two clocks with this converter.

```
models/environment/time/verif/SIM_5_all_inclusive/S_define
120       P_TIME  ("initialization") manager.register_generic_converter(
121           tai_metveh1,
122           "TAI",
123           "met_veh1");
```

3. Bring in the default data (when it exists)

```
lib/jeod/JEOD_S_modules/Base/jeod_time.sm
127        ("default_data") time_converter_tai_utc_default_data.initialize (
128                                                   &time_converter_tai_utc);
...
135        ("default_data") time_converter_tai_ut1_default_data.initialize (
136                                                   &time_converter_tai_ut1);
```

4. Initialize the TimeManager

```
lib/jeod/JEOD_S_modules/Base/jeod_time.sm
 169       P_TIME  ("initialization") time_manager.initialize ( &time_manager_init);
```

Note 1 – the order in which the clocks, converters, and default data are initialized is not important, although we tend to keep the clocks and converters in the same order as that in which they were declared for ease of reading. However, the initialization of the TimeManager should only be performed AFTER all of the clocks and converters have been registered.

Note 2 – all initialization calls associated with setting up the time simulation-object are given P_TIME priority.

B.2.2.c Runtime Implementation

For many situations, there is only one call, regardless of the complexity of the clock arrangement (note – in many situations, even this call is unnecessary because it is made by other models as needed; however, its inclusion here is strongly recommended insurance because its processing cost is so very minimal). This call should be made at some rate such that there is an integer number of such calls between major models being updated (typically at the *DYNAMICS* rate). In simple simulations, that means calling it at the same *DYNAMICS* rate as the other models.

```
lib/jeod/JEOD_S_modules/Base/jeod_time.sm
 188       (DYNAMICS, "environment") time_manager.update ( exec_get_sim_time());
```

The argument to this function call is the time maintained by the simulation engine for the purpose of scheduling function calls and events.

Note that this will only update the decimal clock representation, not any calendar representation. The conversion to a calendar is comparatively slow, and usually required much less frequently so is not included in the default update. Consequently, for simulations in which a calendar time is required, additional update calls must be made, one per calendar (a calendar representation is something like *August 3, 2012 at 12:34:56.78*, whereas a decimal clock takes a format such as *1234.5678 days past epoch*).

```
lib/jeod/JEOD_S_modules/time_TAI_UTC_UT1_calendar.sm
177       (TIME_CALENDAR_UPDATE_INTERVAL, "environment") time_ut1.calendar_update (
178           exec_get_sim_time());
182       (TIME_CALENDAR_UPDATE_INTERVAL, "environment") time_utc.calendar_update (
183           exec_get_sim_time());
```

B.2.3 Setting the Input File Values

B.2.3.a Defining the Simulation Start Time

The simulation start time can be defined in any declared clock. The defining time is called the **initializer,** its initial value can be defined using one of several representations:

- days since epoch (epoch is clock-specific, often J2000)
- seconds since epoch (epoch is clock-specific, often J2000)
- calendar / clock (depending on clock type)
- Truncated Julian (epoch is pre-Apollo)
- Modified Julian Date (not recommended due to loss of precision, epoch is year ~1900)
- Julian Date (not recommended due to loss of precision, epoch is millennia ago)

```
verif/SIM_dyncomp/Modified_data/date_n_time/11Nov2007.py
    19   jeod_time.manager_init.initializer = "UTC"
    20   jeod_time.manager_init.sim_start_format = trick.TimeEnum.calendar
    ...
    24   jeod_time.utc.calendar_year   = 2007
    25   jeod_time.utc.calendar_month  =   11
    26   jeod_time.utc.calendar_day    =   20
    27   jeod_time.utc.calendar_hour   =    0
    28   jeod_time.utc.calendar_minute =    0
    29   jeod_time.utc.calendar_second =  0.0
```

NOTES:

1. When using the standard JEOD_S-modules, all clock names including a prefix *time_*. An equivalent to this example for a SIM_dyncomp built from standard S_modules woutl be:

```
19   jeod_time.manager_init.initializer = "UTC"
20   jeod_time.manager_init.sim_start_format = trick.TimeEnum.calendar
...
24   jeod_time.time_utc.calendar_year   = 2007
25   jeod_time.time_utc.calendar_month  =   11
26   jeod_time.time_utc.calendar_day    =   20
27   jeod_time.time_utc.calendar_hour   =    0
28   jeod_time.time_utc.calendar_minute =    0
29   jeod_time.time_utc.calendar_second =  0.0
```

2. line 20 in the illustrating examples above demonstrates the use of enumerated lists. The prefix "*trick.*" on the assignment is to access the enumerated list defined in *TimeEnum*. This will be seen multiple time during the course.

B.2.3.b Defining the Clock Dependencies

This is an optional, but recommended step; the code has a search algorithm to identify the dependencies but it is safer to specify them.

As each clock updates, it must pull a reference time from some other clock. Clock dependencies specify to which clock a particular clock should look for getting its updates.

There are actually two trees to define, one for initialization and one for runtime.

- The initialization tree must ultimately trace back to the **initializer**.
- The update tree must ultimately trace back to dynamic time (from where it ultimately connects to the simulation engine clock/counter). Dynamic time is given the name "Dyn"; this is needed for constructing the update tree.

The converters that provide for each dependency must be declared and registered with the TimeManager.

NOTE – the initialization tree will typically contain one fewer assignment than the update tree because the **intializer** has no parent in the initialization tree.

verif/SIM_dyncomp/Modified_data/date_n_time/11Nov2007.py

```
31   jeod_time.tai.initialize_from_name = "UTC"
32   jeod_time.ut1.initialize_from_name = "TAI"
33   jeod_time.tt.initialize_from_name  = "TAI"
34   jeod_time.gmst.initialize_from_name  = "UT1"
35
36   jeod_time.tai.update_from_name = "Dyn"
37   jeod_time.ut1.update_from_name = "TAI"
38   jeod_time.utc.update_from_name = "TAI"
39   jeod_time.tt.update_from_name  = "TAI"
40   jeod_time.gmst.update_from_name = "UT1"
```

B.2.3.c Defining Epochs for User-Defined Times (optional)

When defining a clock, its epoch must be set relative to some reference clock. The reference clock could be dynamic time, a standard clock, or another (already anchored) user-defined clock. The process requires either

1. specifying the initial value of the clock (with the simulation-start already anchored, this anchors the clock), or

2. specifying the value of the reference clock at this clock's epoch.

The latter process is very similar to that for defining the simulation start time, and is described in detail in the Time Model document. In particular, using a user-defined clock as the initializer (i.e. to initialize the simulation) requires some careful configuration because both the initial value (simulation start) *and* some epoch must be defined.

In this example, the epoch of the time is defined as some time expressed in UTC, and the simulation is started when this clock has value of *50s*. (i.e. at *23:59:50.0* on *Dec 31, 1998*).

models/environment/time/verif/SIM_5_all_inclusive/SET_test/RUN_UDE_initialized/input.py

```
 28 jeod_time.manager_init.initializer = "met_veh1"
 29
 30 jeod_time.metveh1.epoch_defined_in_name = "UTC"
 31 jeod_time.metveh1.epoch_format = trick.TimeEnum.calendar
 32 jeod_time.metveh1.epoch_year = 1998
 33 jeod_time.metveh1.epoch_month = 12
 34 jeod_time.metveh1.epoch_day = 31
 35 jeod_time.metveh1.epoch_hour = 23
 36 jeod_time.metveh1.epoch_minute = 59
 37 jeod_time.metveh1.epoch_second = 0.0
...
 42 jeod_time.metveh1.initial_value_format = trick.TimeEnum.clock
 43 jeod_time.metveh1.clock_day = 0
 44 jeod_time.metveh1.clock_hour = 0
 45 jeod_time.metveh1.clock_minute = 0
 46 jeod_time.metveh1.clock_second = 50.0
```

B.2.3.d Pausing a Mission Elapsed Time (optional)

A rarely used feature is the ability to pause the clock and pick up again some time later where it left off. This is useful in situations such as countdown pause.

Two values need to be set when performing such a hold: the hold must be set/unset, and the manager must be told that it needs to re-calculate the update. The easiest way of doing this is to set *simtime* to 0.

In this example, there is a hold scheduled for the *met_veh2* clock to take place between the values 10 to 20 of simulation time. Any determining factor can be used, what is important here is the two values that are set.

models/environment/time/verif/SIM_5_all_inclusive/SET_test/RUN_UTC_initialized/input.py

```
102 trick.add_read(10,  """
103 jeod_time.metveh2.hold = True
104 jeod_time.manager.simtime = 0
105
106 """)
107
108
109 trick.add_read(20,  """
110 jeod_time.metveh2.hold = False
111 jeod_time.manager.simtime = 0
112
113 """)
```

B.2.3.e Changing the Rate of Time (optional)

Because the TimeManager determines how dynamic time (used in integration) advances with the simulation-time counter, the actual elapsed time between function calls can be altered on-the-fly. Since all function calls

and logging calls are based on the advance of the simulation engine's simulation-time, then if one tick of simulation time is changed to represent, say, 1% of what it did, then the effect will be that the function calls and data logging will be made 100 times more frequently. This allows for easy manipulation to, for example, slow the simulation down at points of interest, or run the simulation backward from some known configuration.

The advance rate is controlled by the variable *scale_factor*, found in the *TimeDyn* class.

- *scale_factor* < *0*: time runs backwards
- *0* < *scale_factor* < *1*: time advances slowly (more frequent job calls)
- *1* < *scale_factor*: time advances quickly (less frequent job calls)

models/environment/time/verif/SIM_2_dyn_plus_STD/SET_test/RUN_scale_factor_changes/input.py

```
 48 trick.add_read(5,  """
 49 jeod_time.time_manager.dyn_time.scale_factor = -1.0
 50 """)
 51
 52 trick.add_read(10,  """
 53 jeod_time.time_manager.dyn_time.scale_factor = 0.5
 54 """)
 55
 56 trick.add_read(20,  """
 57 jeod_time.time_manager.dyn_time.scale_factor = -2
 58 """)
```

WARNING - the default behavior in Trick's trick_qp application plots the Trick simulation-time *(sys.exec.out.time)* on the *x*-axis. If the scale-factor is manipulated, be sure to plot with JEOD's dynamic time on the *x*-axis instead.

B.2.4 Logging Data

Typically the only value logged from the Time Model is the dynamic time, and then only when it differs from the simulation time. Occasionally, clock or calendar values are also logged since these are not trivially derived from the simulation time.

models/environment/time/verif/SIM_5_all_inclusive/Log_data/log_rec.py

```
 27   dr_group.add_variable( "jeod_time.ut1.calendar_second")
 28   dr_group.add_variable( "jeod_time.utc.calendar_second")
 29   dr_group.add_variable( "jeod_time.manager.dyn_time.seconds")
```

B.3 Dynamics Manager

Document reference: *models/dynamics/dyn_manager/docs/dyn_manager.pdf*

The Dynamics Manager provides the heart of JEOD operations. It is included second in this section because it often appears second in the *S_define* because it requires an input from the Time Model. It must be included in all simulations. In the standard JEOD S_modules, we include two different implementations of the Dynamics Manager:

1. ***dynamics.sm*** is the most frequently used sim-module. It includes the integration-class job-call to perform the integration in the case that there is only one integration group (default operation, for more information on integration groups, see section I.4).

2. ***dynamics_init_only.sm*** only includes the initialization process. It has no integration calls nor environment update calls, so is suitable for unit-tests.

3. ***dynamics_multi_group.sm*** also omits the integration-class job-call, but does include ephemerides and events calls. It is intended to be used to support simulations involving multiple-integration-groups (see section I.4).

B.3.1 What the Dynamics Manager Does

B.3.1.a Manages the Reference Frame Tree

We will see much more about reference frames and trees in chapter D. In short, a reference frame includes a set of coordinate-axes relative to which a vehicle's state can be expressed. All reference frames are positioned and oriented with respect to every other reference frame. If the state is known in one, it can, in principle, be calculated in any other. The Dynamics Manager manages the relative positioning and orientation of the frames by placing all frames in a tree structure.

B.3.1.b Manages the Dynamic Elements of a Simulation

There are two classes of dynamic elements in the simulation – those whose state must be integrated because its course through time is unknown (such as a vehicle), and those whose course is pre-determined (such as a planet).

The Dynamics Manager manages both types of elements, providing:

- integration of translational and rotational states,
- the capability to modify the configuration or trajectory of the body,
- ephemeris models for translational states of planetary bodies, and
- somewhat *ad hoc* models for rotational states of the same.

B.3.1.c Maintains Registries of all Dynamic Objects

All dynamic objects must be registered with the Dynamics Manager, which maintains that registry. All registries are searchable by name, so a vehicle could find the gravitational field of a planet named “Earth”, for example.

B.3.1.d Initializes the Simulation

Initialization within the Dynamics Manager is one of the more commonly user-encountered problems in setting up the *S_define*. The Dynamics Manager model must itself be initialized fairly early in the initialization process, then the Dynamics Manager model can initialize the simulation, but that happens fairly late in the process. The consequence is that there are two initialization calls, which look very similar and appear in a similar place in the *S_define*, but which perform very different tasks and are performed at very different times.

```
lib/jeod/JEOD_S_modules/Base/dynamics.sm
 38       P_MNGR  ("initialization") dyn_manager.initialize_model ( dyn_manager_init,
 39                                                                  time_manager);
 40       P_MNGR_INIT_SIM ("initialization") dyn_manager.initialize_simulation ( );
```

B.3.2 Setting up the S_define

B.3.2.a Simulation Object Structure

The Dynamics Manager requires input from the Time Manager. We first make a private reference instance of type *TimeManager* in the *DynamicSimObject* class (see the section “Structuring a Simulation into Simulation Objects” (A.4.4) for information on passing data between simulation objects):

```
lib/jeod/JEOD_S_modules/Base/dynamics.sm
 23 protected:
 24    // Reference to the external time manager.
 25    jeod::TimeManager & time_manager;
```

Next, make the constructor for the *DynamicsSimObject* class take an reference argument to the instance of *TimeManager* that was created in the *JeodTimeSimObject* instantiation, and in the initialization list populate the local reference with that passed in:

```
lib/jeod/JEOD_S_modules/Base/dynamics.sm
 30    // Constructor
 31    DynamicsSimObject ( jeod::TimeManager & time_manager_in)
 32    :
 33       time_manager (time_manager_in)
```

Finally, make the instantiation call with the appropriate argument:

```
lib/jeod/JEOD_S_modules/dynamics.sm
 2 DynamicsSimObject dynamics (jeod_time.time_manager);
```

B.3.2.b Dynamics Declarations

There are only two declarations needed – one for the Dynamics Manager, and one for its initializer. It is also advisable to include the BodyAction header at this time, even though it will not be used immediately. Note that the DynManagerInit class is only used during initialization.

lib/jeod/JEOD_S_modules/Base/dynamics.sm

```
17 ##include "dynamics/dyn_manager/include/dyn_manager.hh"
18 ##include "dynamics/dyn_manager/include/dyn_manager_init.hh"
...
21 class DynamicsSimObject: public Trick::SimObject {
...
26 public:
27    jeod::DynManager             dyn_manager;
28    jeod::DynManagerInit         dyn_manager_init;
```

B.3.2.c Initializations with the Dynamics Manager

There are two initialization methods, one to initialize the Dynamics Manager, and one for the Dynamics Manager to initialize the simulation. Note that these two calls are made with different priorities.

lib/jeod/JEOD_S_modules/Base/dynamics.sm

```
38       P_MNGR  ("initialization") dyn_manager.initialize_model ( dyn_manager_init,
39                                                                 time_manager);
40       P_MNGR_INIT_SIM ("initialization") dyn_manager.initialize_simulation ( );
```

Important Note – the correct sequencing of these two functions in the overall initialization process is critically important.

The first call (*initialize_model*) must be made before any environment initializations (excluding time, which is usually given its own priority) are made. In *lib/jeod/JEOD_S_modules/dynamics.sm*, that first call is given a *P_MNGR* priority.

The second call (*initialize_simulation*) must be made after initialization of all planetary and vehicular objects but before any state information is provided, including vehicular states and derived states. It is given a *P_MNGR_INIT_SIM* priority.

Notes:

1. initialization of a vehicle (not including its state) is conducted as part of the environment (*P_ENV*, see *vehicle-basic.sm*)

2. because of the settings in *default_priority_settings.sm*, all of the initialization-class jobs tagged with either *ENV* (environment) or *EPH* (ephemerides) will be executed between these two function calls, even though they appear next to one other in the *S-module*.

```
lib/jeod/JEOD_S_modules/default_priority_settings.sm
 2 #define P_MNGR  P20   // Dynamics manager initialization
 3 #define P_ENV   P30   // Environment initializations
 4 #define P_EPH   P35   // Environment initializations
 5 #define P_MNGR_INIT_SIM P39 // Reserved for DynManager::initialize_simulation.
```

Care must be taken to get this right or the simulation will not work, and the error messages may be very abstract. Not all simulations in the JEOD code-base can be relied upon as having a structure with universal applicability.

B.3.2.d Dynamics Runtime Implementation

There are two optional function calls that appear in *dynamics.sm:* one to *update_ephemerides*, and one to *perform_actions.*

In general, it is recommended to call both. The first *(update_ephemerides)* updates the planetary positions from the Ephemeris Model. Obviously, for simple simulations with no implementation of ephemerides, this call is not necessary. The second *(perform_actions)* is a call associated with the *BodyAction* model, which we will study later.

Very briefly, all actions on a body (e.g. stage separation) can be scheduled with a *BodyAction* instance, and all instances of *BodyAction* (there are typically many in even simple simulations) are registered with the Dynamics Manager. The *BodyAction* model provides a flag to each instance to indicate whether the *BodyAction* is ready to be processed. There are really two different paradigms for handling these; in one, when the *BodyAction* is ready, it gets called directly. In the second, the Dynamics Manager repeatedly cycles through the list of *BodyActions* looking for any that are newly ready for processing, and processes those that are. The *perform_actions* method is used to cycle through the list, looking for actions that are ready for application. By adding the *perform_actions* call to the S-module, the user need only be concerned with setting the appropriate flag to indicate readiness for application; there is no need for the user to also call the specific *BodyAction* instance if it is known by the *DynamicsManager* and the *perform_actions()* method is already scheduled.

B.3.2.e Integration

Most of the JEOD integration capability was moved from being a component of JEOD to its own project, known as ER&-Utilities, or *er7-utils*. The ER&-Utilities package can be acquired independently, or with JEOD, and is also distributed with the Trick simulation engine. Reference may be made in this document to the JEOD integration model (which provides an interface to ER7-Utilities integration, as well as providing some additional capabilities) and to the ER7-Utilities model itself.

We will cover integration options and setting up the integrator in sections B.4, and again in more detail in section I.4. This subsection only illustrates the necessary function calls. If the simulation is running JEOD (or *er7-utils*) integration, there are three calls that need to be made, two derivative-class jobs and one integration-class job.

```
lib/jeod/JEOD_S_modules/dynamics.sm
 52       P_GRAV ("derivative") dyn_manager.gravitation ( );
 53       ("derivative") dyn_manager.compute_derivatives ( );
 54
 55       //
 56       // Integration jobs
 57       //
 58       ("integration", &dyn_manager.sim_integrator) trick_ret =
 59       dyn_manager.integrate ( exec_get_sim_time(),
 60                               time_manager);
```

NOTE - if JEOD is performing the integration tasks all within one group, then this is the only integration job that should be called for any JEOD code. Everything is registered with the Dynamics Manager, which takes care of all integrable objects.

B.3.3 Setting the Input File Values

B.3.3.a DynManagerInit *class*

The Dynamics Manager initializer contains (only) an Operating Mode, the name of the central point *(central_point_name)* and information on the integrators (see section B.4 for setting up integration).

Operating Mode

There are three operating modes that dictate the complexity of the simulation:

1. Empty Space *(EphemerisMode_EmptySpace).*
    - This is the simplest case, and is particularly useful for unit-tests. For an example, *see models/dynamics/derived_state/verif/SIM_Relative.*
    - There is no environment, no gravity, and there are no planets to maintain.
    - There is only one frame in which the integration can be performed; the name of that frame is specified with the *central_point_name* variable.
2. Single Planet *(EphemerisMode_SinglePlanet).*
    - This provides a relatively crude simulation, suitable for propagation of a vehicle in low-orbit where gravitational perturbations from other bodies are going to be negligible. This is also useful for model-level testing where models require the existence of a planet. For example, see *models/dynamics/derived_state/verif/SIM_LvlhRelative.*
    - There is only one frame in which the integration can be performed.
3. Ephemeris-driven *(EphemerisMode_Ephemerides*)
    - This is the default mode
    - It provides detailed simulations.
    - There are multiple potential integration frames, the user must specify.

Central point name

This value is only necessary in *EmptySpace* and *SinglePlanet* modes. In *EmptySpace* mode, the name is entirely to the users discretion, we typically use *Space.* In *SinglePlanet* mode, it must be the name of the planet in the simulation (although this does not have to be one of the known planets because the Ephemerides model will be useless in this mode). In the *Ephemerides* mode, all of the planets are already named so the *central_point_name* it is not needed.

```
models/dynamics/derived_state/verif/SIM_Relative/input_common.py:
 3 dynamics.manager_init.mode = trick.DynManagerInit.EphemerisMode_EmptySpace
 4 dynamics.manager_init.central_point_name = "Space"
```

```
models/dynamics/derived_state/verif/SIM_LvlhRelative/SET_test_val/RUN_test/input.py:
27 dynamics.manager_init.mode = trick.DynManagerInit.EphemerisMode_SinglePlanet
28 dynamics.manager_init.central_point_name = "ref_planet"
…
34 dynamics.reference_planet.name = "ref_planet"
```

```
(conceptual):
 45 dynamics.manager_init.mode = trick.DynManagerInit.EphemerisMode_Ephemerides
```

NOTES:

1. Once again we see the required process for accessing enumerated lists. The prefix *trick.* on the assignment in lines 35 and 45 above is to access the enumerated list defined in *DynManagerInit.* This will be seen multiple time during the course.
2. When we look at adding vehicles, we will see that their integration frame must be specified. In EmptySpace and SinglePlanet modes, this frame should be "*x.inertial*", where *x* is the central point name.

```
models/dynamics/derived_state/verif/SIM_Relative/SET_test/RUN_no_rot_B_trans/input.py:
 54 vehicleA.body.integ_frame_name = "Space.inertial"

models/dynamics/derived_state/verif/SIM_LvlhRelative/SET_test_val/RUN_test/input.py:
 52 vehicleA.body.integ_frame_name = "ref_planet.inertial"
```

3. Because EphemerisMode_Ephemerides is default, it would be unusual to see this actually specified in an input file.

B.3.3.b Populating the Dynamics Manager

The Dynamics Manager must be made aware of the following:

- Integration Options
  - Set in the DynManagerInit class
  - See Section B.4 for details
- Vehicular Bodies
  - Registered with the *initialize_model* method
  - See section B.5 for details
- Planetary Bodies

  - Registered with the *register_model* method
  - See Section C.2 for details
- Gravity
  - Registered with the *initialize_model* method
  - See Sections C.2 and C.4 for details
- Instances of actions on vehicular bodies (Body Actions)
  - The list of body actions is NOT ordered
  - Each is registered with the *add_body_action* method
  - Method called from input file, not from the S_define
  - See Chapter F for details
- Derived States (optional registration)
  - Registration from within the *initialize* method
  - See Chapter H for details

B.4 Integration

Document reference: *models/utils/integration/docs/integration.pdf*

The integration process is defined in the Dynamics Manager, but is broken out here for easier navigation. At this point, we will consider setting up only basic integration; advanced integration topics including integration loops, integration groups, and integrable objects are covered in more detail in Section I.4.

B.4.1 JEOD Integration Concepts

B.4.1.a Integration Technique

JEOD and ER7-Utilities provide several integration techniques, each with various situational-dependent advantages and disadvantages. For details on the techniques available and when they should be used, review the Integration Model documentation (models/utils/integration/docs/integration.pdf). The JEOD Integration Model is the JEOD-interface to ER7-Utilities integration models.

A good multi-purpose integrator is the RK4 method. We will be using this one through most of the course.

The techniques available in the ER7-Utilities model are:

- ABM4 (Adams-Bashforth-Moulton 4^{th} order)
- Beeman
- Euler
- MM4 (Modified-Midpoint)
- NL2 (Nystrom-Lear 2^{nd} order)
- Position Verlet
- Velocity Verlet
- RK2-Heun (Heun's method)
- RK2-Midpoint (Runge-Kutta 2^{nd} order midpoint)
- RK4 (Runge-Kutta 4^{th} order)
- RKF45 (Runge-Kutta-Fehlberg $4^{th}/5^{th}$ order)
- RKF78 (Runge-Kutta-Fehlberg $7^{th}/8^{th}$ order)
- RKG4 (Runge-Kutta-Gill)
- RKN4 (Runge-Kutta-Nystrom)
- Symplectic Euler

Ina ddition, JEOD provides the following long-arc integrators; these typically have substantial initialization cost but are very useful for rapid propagation across large expanses of free-coasting flight, e.g. long transfer orbits, or for orbital stability analyses.

- Gauss-Jackson
- LSODE (Livermore Solver)

B.4.1.b Integration Group

All integrable objects are allocated to a group depending on the chosen integration parameters. For simple simulations, all objects will be integrated at the same rate and using the same technique. Thus, there will be only 1 group. Setting up the integrators for multiple groups is covered in section I.4.2.

B.4.2 S_define setup

The Dynamics Manager must be informed of the desired integration options. There are three methods that are commonly found in legacy code, but only one of these is recommended. We will cover only this recommended method in this course. All three methods are mentioned in section 4.2.2 in the Integration Model document.

The first step is to include the header file for all desired integrators; each integration technique has a header file called <*technique*>*_integrator_constructor.hh*. For users writing their own Dynamics simulation object, the most appropriate place to include those headers would be in that object. If using one of the JEOD-provided simulation-modules for the Dynamics simulation object (e.g. *dynamics.sm*), the most appropriate place may be right before the inclusion of that simulation object. However, they may be included anywhere in the S_define outside a class-definition block of code as shown in the next example.

models/utils/integration/verif/SIM_GJ_test/S_define

```
 26 #include "dynamics.sm"
...
 36 ##include "utils/integration/gauss_jackson/include/gauss_jackson_integrator_constructor.hh"
 37 ##include "er7_utils/integration/rk4/include/rk4_integrator_constructor.hh"
```

The next step comes on the last line of the S_define and establishes the integration. The preferred form is:

- IntegLoop sim_integ_loop (*integration_rate*) <*simulation object list*>

Note that *integration_rate* specifies the frequency with which these methods are called. This is a Trick-engine call, so it represents the simulation time, not the dynamic time. Conversely, the integration itself uses dynamic time as the independent variable (so while integration jobs may process every tick of simulation-time, each such tick may be unequal to a dynamic second). While these two time types are configured to be equivalent by default, they can be made to run at different rates. See section B.2.3.e for the distinction between simulation time and dynamic time if it is desirable to integrate at variable rates.

<*simulation object list*> is a comma-separated list of all of the simulation objects that need integrating.

models/utils/integration/verif/SIM_GJ_test/S_define

```
 92 IntegLoop sim_integ_loop (DYNAMICS) dynamics;
```

```
verif/SIM_dyncomp/S_define
677 IntegLoop sim_integ_loop (DYNAMICS) earth, sv_dyn, dynamics;
```

B.4.3 Making and Assigning the Integrator-Constructor

The appropriate integrator constructor for the desired integration technique must now be instantiated and passed to the Dynamics Manager via its initializing class (*DynManagerInit*). This can be handled in the S_define (or an associated sim-module), or in the input file.

B.4.3.a Input file Configuration

With the appropriate integrator constructor header-file included in the S_define, it is possible to construct an instance of that integrator in the input file, and assign that to the *integ_constructor* variable in the *DynManagerInit* class.

We prefix the class constructor with *trick* in order to access the constructor (which is only known to the simulation engine at this time).

```
##include "utils/integration/include/rk4_integrator_constructor.hh"
```

```
models/utils/integration/verif/SIM_GJ_test/SET_test/RUN_RK4_step0/input.py
  3 rk_integrator = trick.RK4IntegratorConstructor()
...
 15 dynamics.dyn_manager_init.integ_constructor = rk_integrator
```

In this example, the integration will now take place using an RK4 integration algorithm.

NOTE – the variable *integ_constructor* is a pointer to the actual instance, but Python will identify that and set *integ_constructor* to the address of *rk_integrator*.

B.4.3.b S_define/input Combination

The second implementation of this preferred method is to instantiate the integrator constructor in the S_define, and then assign it to the *integ_constructor* in the input file. Remember to include the appropriate header. The instance type should be prefaced with *"er7_utils::"* to provide the appropriate namespace for this data type.

```
##include "utils/integration/include/rk4_integrator_constructor.hh"
class DynamicsSimObject: public Trick::SimObject {
...
DynManagerInit           dyn_manager_init;
er7_utils::RK4IntegratorConstructor rk4_integrator_constructor;
...
}
DynamicsSimObject dynamics( ... )
```

```
dynamics.dyn_manager_init.integ_constructor = dynamics.rk4_integrator_constructor
```

B.4.3.c S_define only

The third implementation is handled entirely in the S_define. Assign the value to the manager-initializer in the constructor of the sim-object. This makes it impossible to switch integrators without recompiling and provides no significant advantages, so is not a commonly used technique.

```
##include "utils/integration/include/rk4_integrator_constructor.hh"
class DynamicsSimObject: public Trick::SimObject {
  ...
  jeod::DynManagerInit           dyn_manager_init;
  er7_utils::RK4IntegratorConstructor rk4_integrator_constructor;
  ...
  DynamicsSimObject( <arguments>)
    :
    <optional initializer list>
  {
    dyn_manager_init.integ_constructor = &rk4_integrator_constructor.hh;
    ...
    < continue with job-specifications >
    ...
  }
};
DynamicsSimObject dynamics( ... );
```

Exercise 4. Identifying the Integration Algorithm

B.5 Vehicle

Document reference: ***models/dynamics/dyn_body/docs/dyn_body.pdf***

models/dynamics/body_action/docs/body_action.pdf

Each vehicle is typically given its own S_define object. In that object, we need to construct a representation of the dynamic entity, and optionally associate other peripheral features to the vehicle. At this point in the course, we are only concerned with the fundamentals. We will add to this as the course progresses.

B.5.1 Important concepts

The dynamic entity that is used to represent a vehicle is a *DynBody*. *DynBody* has particular methods for integrating its 6-degree state (3 translational, 3 rotational). The *DynBody* class has a *MassBody* (among other things) and adds state information that a *MassBody* does not have.

That is not to say that a *DynBody* has a state, this is an important realization. In a strict interpretation, vehicles do not have a state per se, but have reference frames, which have state. In the *DynBody* that we use to represent the vehicles, there are three important reference frames: *structure*, *composite-body*, and *core-body*. We will look at reference frames and what they mean in Section D.1.

B.5.2 S_define setup

B.5.2.a Body Declarations

Declare instances of the body, and the processes by which its state and properties are initialized. There are multiple options for initializing state, and we will see more of this when we look at the *BodyAction* model in detail in Chapter F. For now, we will use the basic translational and rotational state initializations.

```
lib/jeod/JEOD_S_modules/vehicle_basic.sm
  8 ##include "dynamics/dyn_body/include/dyn_body.hh"
...
 10 ##include "dynamics/body_action/include/dyn_body_init_trans_state.hh"
 11 ##include "dynamics/body_action/include/dyn_body_init_rot_state.hh"
 12 ##include "dynamics/body_action/include/mass_body_init.hh"
...
 21    jeod::DynBody     dyn_body;
...
 24    jeod::DynBodyInitTransState   trans_init;
 25    jeod::DynBodyInitRotState     rot_init;
 26    jeod::MassBodyInit            mass_init;
```

B.5.2.b Body Initialization

The DynBody is initialized and registered with the Dynamics Manager with the *DynBody::initialize_model* call.

```
lib/jeod/JEOD_S_modules/vehicle_basic.sm
 32       P_ENV ("initialization") dyn_body.initialize_model(dyn_manager );
```

This job must be called between the two initialization jobs found in the Dynamics Manager simulation object.

B.5.2.c Dynamic Effects on Vehicle

There are four sources of dynamic effects:

- Gravitational acceleration
    - Handled by the dynamics manager
- Environmental forces and torques

- Including aerodynamic drag, etc.

- Effector Forces and Torques (e.g. thrusters)
  - The effectors are not modeled within JEOD, but the resulting forces and torques generated by effectors can be incorporated for integrating the state.

- Non-transmitted forces and torques
  - These are forces and torques applied to a vehicle that are **not** transmitted to the vehicle's parent.

Note – Environmental and Effector effects are covered in much more detail in chapter G.

Effector and Environmental effects are calculated separately and added together to give overall forces and torques. Remember not to count something in both categories, or it will be double-counted!

An example of non-transmitted forces and torques would be gravity gradient torque. This is calculated based on the composite inertia of the vehicle. Since composite inertia already includes the inertia of all "child" bodies, adding the torque from those child bodies would result in double-counting. Instead, if a body is a child (not root), its gravity gradient torque will NOT be counted separately.

Forces and torques have to be collected, then superposed, converted into an acceleration, combined with gravitational acceleration, and integrated into the translational and rotational velocities

Forces and torques are always expressed in Structural frame

Forces and torques may be updated as environment or derivative class jobs

Forces and torques are collected using the *vcollect* method:

```
concept, based on verif/SIM_dyncomp/S_define:
677   vcollect sv_dyn.body.collect.collect_effector_forc jeod::CollectForce::create {
678      vehicle.force_extern
679   };
680   vcollect sv_dyn.body.collect.collect_environ_forc jeod::CollectForce::create {
        vehicle.rad_pressure.force,
        vehicle.aero_drag.aero_force
681   };
685   vcollect sv_dyn.body.collect.collect_effector_torq jeod::CollectTorque::create {
686     vehicle.torque_extern
687   };
688   vcollect sv_dyn.body.collect.collect_environ_torq jeod::CollectTorque::create {
        vehicle.rad_pressure.torque,
        vehicle.aero_drag.aero_torque,
689   };
690   vcollect sv_dyn.body.collect.collect_no_xmit_torq jeod::CollectTorque::create {
692     vehicle.grav_torque.torque
693   };
```

In this example, force_extern and torque_extern are (previously declared) variables representing external effector forces. They are collected into the effector categories. Aerodynamic force and torque are collected into the environment forces and torques. Gravity gradient torque is collected into the non-transmitted torque category.

Notes:

- Gravitational force is never collected, because it is handled by the Dynamics Manager as an acceleration that gets added to the resulting acceleration from the forces collected here.
- In this example, there is no non-transmitted force, so that collection is omitted.
- For root bodies, it does not matter whether the forces/torques are collected as transmittable or non-transmittable quantities. The only distinction is in whether those effects get propagated to the parent. In SIM_dyncomp, aero_force is included as a non-transmitted force/torque, which is typically incorrect. But because it is a root-body, there is no consequence.

B.5.3 Setting the Input File Values

Most of the intialization process takes place in the input file, and is associated with setting up the *BodyAction* initializers correctly. Details on the *BodyAction* model are found in Chapter F; we will consider the basics here.

B.5.3.a Defining Vehicle Basic Properties

The following properties must be set:

- Name of vehicle
- Integration frame (should be inertial)
- Whether to integrate translational state
- Whether to integrate rotational state

```
verif/SIM_dyncomp/SET_test/RUN_1/input.py
  113 sv_dyn.body.name = "iss"
  114
  115 #/* Configure vehicle integration information. */
  116 sv_dyn.body.integ_frame_name = "Earth.inertial"
  117 sv_dyn.body.translational_dynamics = True
  118 sv_dyn.body.rotational_dynamics = True
```

NOTE - the *Earth.inertial* assignment referes to the Earth-centered inertial reference frame. For reference frame naming convention, see Section D.1 on Reference Frames.

B.5.3.b Defining Vehicle Mass Properties

Almost universally, we use the *MassBodyInit BodyAction* to define mass properties (again, see Chapter F for details on the *BodyAction* model and section F.4 specifically for details of using the *MassBodyInit BodyAction*).

At this point, we will cover only the basic steps necessary to get a sim up and running. The following values should be set to define mass:

- Identification of the subject that is being initialized

```
verif/SIM_dyncomp/Modified_data/mass/iss.py
    13   #/* Set the mass porperties for this vehicle. */
    14   veh_obj_reference.mass_init.subject = veh_obj_reference.body.mass
```

- The mass

```
verif/SIM_dyncomp/Modified_data/mass/iss.py
   24    veh_obj_reference.mass_init.properties.mass = trick.attach_units( "kg",400000.0)
```

  Note – All JEOD variables are handled as SI units. In this case, the units specification is not necessary because the value will default to kg anyway. But it is included for completeness.

- The position of the center of mass (in structural reference frame)

```
verif/SIM_dyncomp/Modified_data/mass/iss.py (modified)
25 veh_obj_reference.mass_init.properties.position = trick.attach_units( "m",[ -3.0,-1.5,4.0])
```

- The Structure-frame-to-Body-frame orientation (see the information on specifying orientation in section I.2 and also the Mass documentation (at *models/dynamics/mass/docs*) for more details)

```
verif/SIM_dyncomp/Modified_data/mass/iss.py (modified)
17 veh_obj_reference.mass_init.properties.pt_orientation.data_source = trick.Orientation.InputEigenRotation
20 veh_obj_reference.mass_init.properties.pt_orientation.eigen_angle  = 0.0
21 veh_obj_reference.mass_init.properties.pt_orientation.eigen_axis    = [ 0.0, 1.0, 0.0]
```

  - As an option, it is possible to specify the body-to-structure orientation instead of Structure-to-Body. In the absence of the command to do so, the default setting is to interpret any orientation as Structure-to-Body

```
(extension based on verif/SIM_dyncomp/Modified_data/mass/iss.py)
   veh_obj_reference.mass_init.properties.pt_frame_spec = trick.MassBasicPointInit.BodyToStruct
```

- The Inertia Tensor

```
verif/SIM_dyncomp/Modified_data/mass/iss.py (modified)
26 veh.mass_init.properties.inertia[0] = trick.attach_units("kg*M2",[ 1.02e+8,-6.96e+6,-5.48e+6])
27 veh.mass_init.properties.inertia[1] = trick.attach_units("kg*M2",[-6.96e+6, 0.91e+8, 5.90e+5])
28 veh.mass_init.properties.inertia[2] = trick.attach_units("kg*M2",[-5.48e+6, 5.90e+5, 1.64e+8])
```

Because the inertia tensor value is frame-dependent, it is necessary to specify a reference frame along with the numerical values of the tensor. By default, this frame is assumed to be the core-body frame, but other options exist, as described in section 4.1.1 of the Mass document *(models/dynamics/mass/docs/mass.pdf)* and section F.4 in this document. The user may specify to use the structure frame, or a combination of structure-frame axis-orientation with body-frame origin, or a manually specified axis orientation and/or origin.

The *MassBodyInit BodyAction* used to define the mass properties must be added to Dynamics Manager list of *BodyAction* instances.

```
verif/SIM_dyncomp/SET_test/RUN_1/input.py
   152 dynamics.manager.add_body_action(sv_dyn.mass_init)
```

B.5.3.c Defining Vehicle Initial State

Vehicles have three reference frames (*composite-body*, *core-body*, and *structure*). The state of any one of these may be specified, but we must specify which one. This is handled with one of several BodyAction options (see Chapter F for details on the *BodyAction* model).

Must specify:

- The subject body
- (optional) Name the body action (used for debugging only)
- The reference reference-frame (relative to which the data is represented)
- The subject reference-frame (of the body) whose state is being represented.
- (optional) Whether to initialize position, velocity, or both (defaults to both)
- The position and/or velocity vector of the subject reference-frame relative to the reference reference-frame, expressed in the reference reference-frame.

```
Conceptual:
sv_dyn.trans_init.dyn_subject      = sv_dyn.body
sv_dyn.trans_init.action_name      = "TRANSLATIONAL_STATE_INIT"
sv_dyn.trans_init.reference_ref_frame_name = "Space.inertial"
sv_dyn.trans_init.body_frame_id      = "composite_body"
sv_dyn.trans_init.position  = trick.sim_services.attach_units("m" ,   [0.0 , 0.0 , 0.0])
sv_dyn.trans_init.velocity  = trick.sim_services.attach_units("m/s" , [1.0 , 0.0 , 0.0])
```

The same is done for the *BodyAction* used to initialize the rotational state (there are multiple *BodyAction* options for specifying both the translational and rotational state, see Chapter F for more details):

```
Conceptual:
sv_dyn.rot_init.dyn_subject         =  sv_dyn.body
sv_dyn.rot_init.action_name       = "ROTATIONAL_STATE_INIT"
sv_dyn.rot_init.reference_ref_frame_name    = "Space.inertial"
sv_dyn.rot_init.body_frame_id       = "composite_body"
sv_dyn.rot_init.orientation.data_source     = trick.Orientation.InputEulerRotation
sv_dyn.rot_init.orientation.euler_sequence  = trick.sim_services.Yaw_Pitch_Roll
sv_dyn.rot_init.orientation.euler_angles = trick.sim_services.attach_units("d",[0.0,0.0,0.0])
sv_dyn.rot_init.ang_velocity        = trick.sim_services.attach_units("d/s" , [0.0,1.0,0.0])
```

Then these *BodyAction* objects are added to the Dynamics Manager list of *BodyAction* instances:

```
Conceptual:
dynamics.manager.add_body_action( sv_dyn.trans_init )
dynamics.manager.add_body_action( sv_dyn.rot_init )
```

B.5.4 Logging Vehicle State

The most commonly logged items include:

- The state of one of the vehicle's frames, usually the composite body frame:

```
example
for ii in range(0,3) :
  dr_group.add_variable("veh.body.composite_body.state.trans.velocity[" + str(ii) + "]")
  dr_group.add_variable("veh.body.composite_body.state.trans.position[" + str(ii) + "]")
  dr_group.add_variable("veh.body.composite_body.state.rot.ang_vel_this[" + str(ii) + "]")
  for jj in range(0,3):
    dr_group.add_variable("veh.body.composite_body.state.rot.T_parent_this["+str(ii)+"]["+str(jj)+"]")
...
```

- The accelerations of the vehicle

```
for ii in range(0,3) :
  dr_group.add_variable("veh.body.derivs.trans_accel[" + str(ii) + "]"
  dr_group.add_variable("veh.body.derivs.rot_accel[" + str(ii) + "]" )
```

In this example, position and velocity are of the *composite_body* frame with respect to and expressed in the inertial integration frame.

Angular velocity is of the desired frame with respect to its parent (inertial in this case), expressed in the desired frame *(composite_body* in this case).

The transformation is from the parent frame to the desired frame.

The *derivs* represent the second derivative of the state of the body frame. So *deriv.trans_accel* is the linear acceleration of the body frame with respect to and expressed in the inertial frame, while *deriv.rot_accel* is the angular acceleration of the body frame with respect to the inertial frame, expressed in the body frame.

Exercise 5. Simple simulation

Exercise 6. Simple Simulation, adding clocks

Exercise 7. Two-vehicle Simulation

Chapter C Building the Environment

C.1 Introduction to Environment

Definition

The *Environment* is that which can passively affect the behavior of a vehicle.

It does not include the vehicle itself (since a vehicle does not affect its own behavior), nor any active effects such as thrusters. It does not include any of the abstract mathematical treatments, such as that provided by the Integration Model, or the Dynamics Manager. It does not include the effect itself, such as Aerodynamic drag.

The major entities in the environment are the gravitational field, the positions of the planets (which produce the gravitational field), any atmosphere, radiation fields, etc.

Typically, we create one simulation object for the environment at the top-level that contains the Ephemeris Model and the Gravity Manager, then one for each planet which contain planet-specific implementations of the Gravity Manager and other planet-specific models.

C.2 Gravity - Starting an Environment Object

Document reference: *models/environment/gravity/docs/gravity.pdf*

C.2.1 Introduction

C.2.1.a Isolating Gravity, the Planet, and the Vehicle.

Gravity is an all-pervading environmental effect, it is also highly dependent upon the individual planets, and it has direct effect on the individual vehicles. There are arguments for placing the gravity model in any one of these three places. In JEOD, we divide the gravity manager into more manageable parts, and provide:

1. A top-level Gravity Manager that typically resides in the environment simulation-object
2. A planet-specific implementation that provides an instance of a gravity source (usually a *SphericalHarmonicsGravitySource* class) that is added to the Gravity Manager. This typically resides in the appropriate planet simulation-object.
3. A vehicle-specific set of controls so that the implementation and resolution of each gravitational field can be set on a vehicle-by-vehicle basis. This typically resides in the appropriate vehicle simulation-object.

The ability to configure gravity specifically to each vehicle allows for optimal performance. For example:

- A vehicle in a very low Earth may require a high resolution Earth gravity model
- A vehicle in a higher orbit may require lower resolution on the Earth field, but perturbations from Sun and Moon

- A vehicle in a lunar orbit may require a high resolution lunar gravitational field, with perturbations from Earth and Sun.

These three can all be accommodated simultaneously in one simulation without the expense of the high-resolution calculations for the vehicles that do not need them.

Each vehicle simulation-object then has the ability to select which gravitational fields to use; each planet simulation-object knows directly only of its own gravity source (not any other gravitational sources, and not any information about which vehicles may – or may not – be in that field), while the environment simulation object knows of the umbrella Gravity Manager, which knows of all of the gravitational bodies in the simulation. The Dynamics Manager ties this all together and knows of all of the vehicles and of the Gravity Manager.

C.2.1.b How the Gravity Manager Works

The computation performed for gravity involves calculating the gradient of the gravitational field, thereby providing an acceleration. The gravitational force, per se, is never actually computed. Each of the gravity-bodies (a.k.a planetary bodies, i.e. those bodies that produce a modeled gravitational field) in the simulation must be named and registered with the Gravity Manager implementation. Once registered, any gravity-source can be deactivated (i.e. its effect will be removed) or reactivated at any point in the simulation.

JEOD will determine, behind the scenes, whether the calculation of the effect of some gravitational body should be treated as a standard Newtonian $\left(\frac{-GM}{r^2}\hat{r}\right)$ acceleration, or as a 3rd body perturbation to another gravity calculation. We will revisit 3rd body perturbations in section D.7 , but we need to cover how reference-frames are treated before this can be explained.

C.2.2 S_define Setup

Two simulation-modules are provided for the environment, *environment.sm* and *environment_sans_de405.sm*. Both have the Gravity Manager; the *environment.sm* file also adds the Ephemerides Model, which is omitted from the *environment_sans_de405.sm* module.

When operating in *EmptySpace* or *SinglePlanet* mode (see Section B.3.3.a for information on operating mode), there is no need to have the *Ephemerides* Model, so using *environment_sans_de405.sm* is more efficient. When operating in Ephemeris-driven mode, the model should be included, so use *environment.sm.*

We will look at *environment_sans_de405.sm* first to set up gravity, then return to *environment.sm* to add the Ephemerides model.

The Gravity Manager setup is actually very straightforward. It is instantiated, and registered with the dynamics manager as a *P_ENV* (environment) priority initialization class job. NOTE – the priority is driven by the requirement that this call comes before the Dynamics Manager *initialization_simulation* call (perhaps *P_BODY*, or *P_MNGR_INIT_SIM*).

There are no update methods to call; the Dynamics Manager takes care of managing the gravity once the simulation is configured and running.

Because the gravity manager must register with the Dynamics Manager, the environment simulation-object must receive the Dynamics Manager as a constructor-input, so must be defined in the S_define after the dynamics simulation-object has been defined, and instantiated after the dynamics simulation-object has been instantiated.

```
lib/jeod/JEOD_S_modules/Base/environment_basic.sm
 10 ##include "environment/gravity/include/gravity_manager.hh"
...
 14 class EnvironmentBasicSimObject: public Trick::SimObject {
...
 20    jeod::GravityManager gravity_manager;
...

 27       P_ENV  ("initialization") gravity_manager.initialize_model ( dyn_manager);
lib/jeod/JEOD_S_modules/environment_sans_de405.sm
 2 EnvironmentBasicSimObject env (dynamics.dyn_manager);
```

We must also make a change to the vehicle simulation-object to add the necessary classes to allow the vehicle-by-vehicle control of the gravitational field resolution. Each vehicle must have a controller for each of the planetary fields of interest to it.

```
Extension of lib/jeod/JEOD_S_modules/vehicle_basic.sm
21 jeod::SphericalHarmonicsGravityControls  earth_grav_control;
   jeod::SphericalHarmonicsGravityControls  moon_grav_control;
   jeod::SphericalHarmonicsGravityControls  sun_grav_control;
```

C.2.3 Setting the Input File values

In the input file, the vehicle gravity controls are assigned to particular planets, and the resolution is defined. In this example, based on the verification simulation, *SIM_Apollo*, we have an earth-orbiting vehicle with 3rd-body perturbations from Sun and Moon. The only gravity gradient calculation is from Earth-gravity.

First, define with which planet the controls are associated (even though the one in this example is called *earth_grav_controls*, the association with Earth is not made until this line.

```
Example, based on sims/SIM_Apollo/Modified_data/vehicle/grav_controls.py:
vehicle.earth_grav_control.source_name = "Earth"
```

Next, set the active flag. This turns this planet's gravity on or off for this particular vehicle.

```
vehicle.earth_grav_control.active      = True
```

Next, decide whether to use the simple spherical gravity, or the more complex spherical harmonics

```
vehicle.earth_grav_control.spherical   = False
```

and if using the spherical harmonics model, set the degree and order

```
vehicle.earth_grav_control.degree      = 8
vehicle.earth_grav_control.order       = 8
```

(optional) Next, consider whether the gravity gradient (gravity gradient produces a torque on asymmetric vehicles, see section C.7) is needed. If desired, also set the degree and order for this calculation.

```
vehicle.earth_grav_control.gradient    = True
vehicle.earth_grav_control.gradient_degree = 8
vehicle.earth_grav_control.gradient_order = 8
```

In the same input file, the other gravity controls are configured in the same way. In this example, that means turning on Moon and Sun, but leaving them spherical and without the gradient calculation in this example.

```
vehicle.moon_grav_control.source_name = "Moon"
vehicle.moon_grav_control.active      = True
vehicle.moon_grav_control.spherical   = True
vehicle.moon_grav_control.gradient    = False
```

Likewise, set the controls for solar gravity.

Finally, add the controls to the Dynamics Body object:

```
vehicle.dyn_body.grav_interaction.add_control(vehicle.earth_grav_control)
vehicle.dyn_body.grav_interaction.add_control(vehicle.moon_grav_control)
vehicle.dyn_body.grav_interaction.add_control(vehicle.sun_grav_control)
```

TIP – In the original example (sims/SIM_Apollo/Modified_data/vehicle/grav_controls.py) the values are specified in a Modified-data-defined Python method. This is a particularly useful technique for multi-vehicle simulations where each vehicle requires the same configuration. By specifying once in a Python method, the same block of code can be executed for every vehicle, with each vehicle passed in one at a time via the argument list and the same values assigned to it.

```
execfile("Modified_data/vehicle/gravity_controls.py")
configure_gravity( vehicleA)
configure_gravity( vehicleB)
```

```
Modified_data/environment/gravity_controls.py (example)
def configure_gravity(vehicle):
  vehicle.earth_grav_control.source_name = "Earth"
  etc.
```

NOTE – a control's values may be changed after it has been added to the Dynamic Body, resulting in direct effect on the field that the vehicle sees. When a control is added to the Dynamic Body, a *reference* to the control gets added, not a *copy* of the control. If it is desirable to change the gravity-field resolution at some point in the sim, it is sufficient to change the value within the control following the same system presented above. That change will be retained until it is changed again or the sim terminates. It is not necessary to re-add the control.

Consequently, to check on a vehicle's current settings for the gravitational field of some planet it is sufficient to check on the values of the Control associated with that vehicle and that planet.

C.3 Planets – A New Simulation-Object

Document reference: *models/environment/planet/docs/planet.pdf*

By *Planet*, we mean any large celestial object capable of exerting a significant gravitational influence. It is not limited to bodies officially designated as planets. We include Sun, Moon, and potentially minor planets and satellites of other planets in this category.

C.3.1 Introduction to Planet

Generally, the purpose of adding a planet is because we want information about its gravitational field (see section C.2), and for that we need information about its position. When we have multiple planets in a simulation, their relative positions are provided by the Ephemerides Model (see section C.6) and the simulation is run in *Ephemerides* mode (see section B.3.3.a for information on operating mode). With a single planet, there is no relative position, no Ephemerides Model to include, and the simulation is run in *SinglePlanet* mode. In *SinglePlanet* mode, remember to name the *central_point_name* (see section B.3.3.a) to the name of the planet. Also, if building a simulation in stages, it is good practice to wait until the planet simulation-object has been fully implemented before making the transition to *SinglePlanet* mode.

The Gravity Manager is not particularly useful without the presence of at least one planet, so this section is intrinsically tied to the previous one. However, because each planet is a physical entity – with specific physical parameters – we typically create a new simulation-object to represent each planet.

The Planet class provides two reference frames – a pseudo-inertial frame *(inertial)* with non-rotating axes, and a planet-fixed frame *(pfix)* with axes that rotate with the planet. The *inertial* reference frame is registered as parent to the *pfix* reference frame. See chapter D for information on reference frames and the associated tree-structure hierarchy, and section H.4 for information specific to the planet-fixed frame.

C.3.2 S_define Setup

There are 5 planet S_modules included in JEOD:

- *earth_basic.sm*
- *earth_GGM02C_MET_RNP.sm*
- *earth_GGM02C.sm*
- *moon_basic.sm*
- *sun_basic.sm*

These can be used as they are, or can easily be adapted to other planets, or extended to include other models. Four of these five are effectively identical, with *earth_GGM02C_MET_RNP.sm* standing out as unique simply because it includes additional models that provide attitude and atmospheric information. These additional models are covered in sections C.5 and G.2, and this S_module will be revisited in those sections.

The other four S_modules have identical structure:

C.3.2.a Planet Declarations

Name the class something intuitive, such as EarthSimObject, and instantiate:

- A Planet class (called *planet in this example)*
- A GravitySource (usually a SphericalHarmonicsGravitySource). In this example, it is named *gravity_source*)
- An instance of the appropriate planetary data class

- An instance of the appropriate gravitational field data class.

```
lib/jeod/JEOD_S_modules/planet_generic.sm
 29    public:
 30     jeod::Planet planet;
 31     jeod::SphericalHarmonicsGravitySource gravity_source;
lib/jeod/JEOD_S_modules/earth_spherical.sm
 16     jeod::Planet_earth_default_data
planet_default_data;
 17     jeod::SphericalHarmonicsGravitySource_earth_spherical_default_data
earth_grav_default_data;
```

The planetary and gravitational data files can be found in *environment/planet/data/* and *environment/gravity/data/* respectively. JEOD includes planet-data for

- Sun,
- Earth,
- Moon,
- Mars, and
- Jupiter,

and gravity data for multiple gravity options (see section C.4 for discussion on the implementation of specific gravity models). For now, recognize that whichever model is being used, the default data for that model must be instantiated and loaded in the *S_define*.

Aside on inheritance: It would be very feasible to create specific Planet objects by inheritance from the Planet class (e.g. create an EarthBasic class) that would then incorporate the specific default data for that specific implementation. If that were done, the instantiation and application of default-data would become superfluous, becoming a component of the default instantiation of the specific planet object. However, to keep the signatures as consistent as possible, the current implementation involves instantiating a generic Planet object and populating it with specific default-data.

Be sure to include the appropriate headers. Since any header need only be included once, the inclusion of headers for classes that are common across multiple planets may be omitted in later simulation-objects (although their redundant inclusion is permitted). Note that in the S_modules, all headers are included for every object so that they may be used independently of one another.

```
lib/jeod/JEOD_S_modules/planet_generic.sm
 15 ##include "environment/planet/include/planet.hh"
 16 ##include "environment/gravity/include/spherical_harmonics_gravity_source.hh"
lib/jeod/JEOD_S_modules/earth_spherical.sm
  8 ##include "environment/planet/data/include/earth.hh"
  9 ##include "environment/gravity/data/include/earth_spherical.hh"
```

C.3.2.b Planet Initialization

The appropriate initialize functions must be called for the default data instances. These functions take arguments to *planet* (for planet-data) or to *gravity_source* (for gravity data).

As *P_ENV* (environment) priority calls, in any order:

- The gravity source must be initialized; this is achieved with the *gravity_source.initialize_body* call.
- The gravity source must be added to the gravity manager; this is achieved with the *grav_manager.add_grav_source* call.
- The planet must be added to the Dynamics Manager, and be associated (both ways) with the gravity source; this is achieved with the *planet.register_model* call.

After the *planet.register_model* call has been made, the *planet.initialize* call can be made. This order is important. In the illustration below, we mark the latter call as *P_EPH* to ensure that this ordering is performed correctly. In this example, *P_ENV* would also work, because the *planet.initialize* call appears later in the file than the *planet.register_model* call. However, relying on file-ordering is potentially dangerous as it assumes the file will not be reorganized at some later time.

- Note 1 – the planet should be initialized before the Dynamics Manager *initialize_simulation* call, so should be made with a *P_EPH* priority.
- Note 2 – to complete these steps, references to the Gravity Manager and Dynamics Manager are required, and must be populated through the constructor (see section A.4.4 for information on simulation-object inter-dependencies).

```
lib/jeod/JEOD_S_modules/earth_basic.sm
 44       ("default_data") planet_default_data.initialize ( &planet );
 45       ("default_data") gravity_default_data.initialize ( &gravity_source );
...
 49       P_ENV  ("initialization") gravity_source.initialize_body ( );
 50       P_ENV  ("initialization") grav_manager.add_grav_source ( gravity_source) ;
 51       P_ENV  ("initialization") planet.register_model ( gravity_source,
 52                                                          dyn_manager);
 53       P_EPH  ("initialization") planet.initialize ( );
...

  1 EarthSphericalSimObject earth( dynamics.dyn_manager, env.gravity_manager);
```

There are no run-time calls to be made for a simple planet implementation that provides only ephemeris and gravity data.

C.3.3 Setting the Input File Values

If using the default data to define the planet, no additional input is required.

Otherwise, the planet must be initialized correctly. It must be named, and if it is to be non-spherical, its shape must be defined. The shape can be defined by specifying the equatorial radius coupled with one of the following:

- Flattening coefficients (or inverse flattening coefficient)
- Ellipsoid eccentricity (or its square)
- The mean polar radius

```
example
earth.planet.name = “Earth”
earth.planet.r_eq = 6378137.0
earth.planet.flat_inv = 298.257223563
```

Exercise 8. Creating a Simple Spherical-Earth Orbiter

Exercise 9. Creating a Simulation with Two Earths

C.4 Selection of Gravity Managers

C.4.1 Introduction

There are several planet-specific gravity managers provided in JEOD, and switching between them is very easy once the initial setup process is understood. The following models are provided:

- the spherical model of Sun gravity,
- the GEMT1 model of Earth gravity,
- the GGM02C model of Earth gravity,
- the spherical model of Earth gravity,
- the LP150Q model of Moon gravity,
- the spherical model of Moon gravity,
- the MRO110B2 model of Mars gravity,
- the spherical model of Mars gravity,
- the spherical model of Jupiter gravity.

Additionally, the GRGM900C model of Moon gravity is available in the JEOD contributions area, but it is not a part of the official JEOD release.

C.4.2 S_define Setup

In the previous section, we looked at the simulation module *earth_basic.sm* which provided a spherical earth gravity model*;* now we will look at the *earth_GGM02C.sm* which provides the GGM02C model, and identify the similarities between the two modules.

In *earth_basic.sm*, we had:

```
lib/jeod/JEOD_S_modules/Base/earth_spherical.sm
  9 ##include "environment/gravity/data/include/earth_spherical.hh"
...
 17     jeod::SphericalHarmonicsGravitySource_earth_spherical_default_data
earth_grav_default_data;
```

In *earth_GGM02C.sm* we have

```
lib/jeod/JEOD_S_modules/Base/earth_GGM02C_baseline.sm
 22 ##include "environment/gravity/data/include/earth_GGM02C.hh"
...
 47    jeod::SphericalHarmonicsGravitySource_earth_GGM02C_default_data
earth_grav_default_data;
```

There is only one difference: the class instantiated for populating the default data (resulting in a change to the class type and the address of the defining header file).

The lesson here is that for any desired model, the implementation is as simple as declaring an instance of the relevant default data class, and calling its *initialize* method with an argument providing the address of the gravity-source associated with that planet.

C.4.3 Setting the Input File Values

The gravity controls input file are more complicated for non-spherical gravity models; these were covered in their full complexity in section C.2.3.

Exercise 10. Complications of Adding a General Gravity Model

C.5 Rotation, Nutation, Precession – Adding to the Planets

Document reference: models/environment/RNP/docs/RNP.pdf

C.5.1 Introduction

If you have completed the exercises to this point, you will already have found the necessity of the RNP model. The trajectory of a vehicle is strongly influenced by the resolution to which we model the gravitational field. For precise work, we must assume a non-spherical gravitational field. But, when the gravitational field becomes asymmetric, it has a defined orientation. It then becomes necessary to be able to model the orientation of the

planet because that determines the orientation of the asymmetric gravitational field, and that determines the trajectory of the vehicle.

The Rotation, Nutation, Precession (RNP) Model can be used to provide that orientation information. It provides orientation based on simple axial rotation, coupled with nutation, precession, and motion (polar motion) of the polar axis. While the inner workings of the model are, in principle, applicable to any planet, JEOD provides data only for implementations to model Mars and Earth. The orientation of Moon is provided by the Ephemerides model, **DO NOT USE THE RNP MODEL FOR LUNAR ORIENTATION**.

JEOD does not provide non-spherical gravity models for any other bodies (apart from Earth, Moon, and Mars), so does not provide orientation data for them. Extending JEOD to provide non-spherical gravity models of other bodies also requires that some means be given to provide the orientation of those bodies.

For the illustrative examples in this section, we will be using the J2000 Earth RNP model. The MarsRNP model is similar, perhaps a little easier to implement. For an example of a simulation implementing MarsRNP, look at *verif/Integrated_Validation/SIM_Mars/S_define.*

C.5.2 Earth-RNP

C.5.2.a S_define Setup

The Earth-RNP model typically resides in the Earth simulation-object.

RNP Declarations

There are 3 classes to instantiate in the simulation-object, each with its additional default-data class:

- one main class (*RNPJ2000*)
- two classes used for initialization (*NutationJ2000Init*, *PolarMotionJ2000Init*)
- three classes used to define default data sets:
  - *RNPJ2000_rnp_j2000_default_data*
  - *NutationJ2000Init_nutation_j2000_default_data*
  - *PolarMotionJ2000Init_xpyp_daily_default_data, or PolarMotionJ2000Init_xpyp_monthy_default_data*

```
models/environment/RNP/RNPJ2000/verif/SIM_RNP_J2000_prop/S_define
 10 ##include "environment/RNP/RNPJ2000/include/rnp_j2000.hh"
 11 ##include "environment/RNP/RNPJ2000/include/nutation_j2000_init.hh"
 12 ##include "environment/RNP/RNPJ2000/include/polar_motion_j2000_init.hh"
...
 35    jeod::RNPJ2000 rnp;
 36    jeod::NutationJ2000Init nut_init;
 37    jeod::PolarMotionJ2000Init pm_init;
```

default-data:

```
models/environment/RNP/RNPJ2000/verif/SIM_RNP_J2000_prop/S_define
 16 ##include "environment/RNP/RNPJ2000/data/include/rnp_j2000.hh"
 17 ##include "environment/RNP/RNPJ2000/data/include/nutation_j2000.hh"
 18 ##include "environment/RNP/RNPJ2000/data/polar_motion/include/xpyp_daily.hh"
...
 41    jeod::RNPJ2000_rnp_j2000_default_data rnp_data;
 42    jeod::NutationJ2000Init_nutation_j2000_default_data nut_data;
 43    jeod::PolarMotionJ2000Init_xpyp_daily_default_data pm_data;
```

Note – the polar motion is interpolated from data points available once-per-month, or once-per day. In this example, the daily option is exercised. See the model documentation for more details.

RNP initialization

The three default data class initialization methods must be processed:

```
models/environment/RNP/RNPJ2000/verif/SIM_RNP_J2000_prop/S_define
 67       ("default_data") rnp_default_data.initialize( &rnp);
 68       ("default_data") nut_init_default_data.initialize( &nut_init);
 69       ("default_data") pm_init_default_data.initialize( &pm_init);
```

Then a sequence of initialization calls, all with P_ENV priority:

The RNP model gets initialized

```
(based on models/environment/RNP/RNPJ2000/verif/SIM_RNP_J2000_prop/S_define)
232       P_ENV ("initialization") rnp.initialize(dyn_mngr.dyn_manager);
```

The Nutation and the Polar Motion components get initialized from their respective initialization classes, which were populated from their respective default data files.

```
 75       P_ENV ("initialization") rnp.nutation->initialize( &nut_init );
 76       P_ENV ("initialization") rnp.polar_motion->initialize( &pm_init );
```

Finally, the run-time call *update* should also be called at initialization to provide initial orientation data.

```
(based on models/environment/RNP/RNPJ2000/verif/SIM_RNP_J2000_prop/S_define)
 77       P_ENV ("initialization") rnp.update_rnp ( tt, gmst, ut1 );
```

Note 1 – the *update_rnp* call requires the output of three clocks. These could be made as individual references and individually passed in as arguments to the constructor, or (in this case) the a reference is made to the entire jeod_time simulation-object, and the individual clocks extracted from that.

Note 2 – the values of the three clocks must be set to their respective times representing simulation initialziation for this call to be successful. The clocks are set to their initial values when the Time Manager initializes the entire tree; as long as that function call has been completed, this call should be good. In the default S_modules, all of the Time Model initialization calls take priority P_TIME, and are completed before any P_ENV calls.

RNP run-time calls

The same *update* function can be called routinely at high frequency, but this is computationally expensive. Since the bulk of the motion is provided by the simple rotation component (rather than the complicated nutation, precession and polar motion components), a simpler method has also been provided, *update_axial_rotation.*

Recommended practice is to run this method relatively frequently, with the full *update* call run less frequently to update the other components.

In the default simulation module, the axial rotation is updated at the derivative rate:

```
lib/jeod/JEOD_S_modules/Base/earth_GGM02C_MET.sm
 95       P_ENV ("derivative") rnp.update_axial_rotation( gmst );
```

while the full update is run at a slower rate:

```
 91       (LOW_RATE_ENV, "environment") rnp.update_rnp ( tt, gmst, ut1 );
```

NOTES:

1. if using the default simulation module, it is necessary to define this second calling rate. It is conventional to collect all of the defined rates together somewhere near the beginning of the S_define.

```
example
#define DYNAMICS 1.0
#define LOW_RATE_ENV 30.0
```

2. Because the earth simulation-object has a derivative-class job, it must be assigned to an integration loop

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

C.5.2.b Setting the Input File Values

The type of RNP update to perform with the *update* function call can be specified in the input file, with options described in section 4.1 of the RNP documentation.

C.5.2.c Logging Data

The attitude of the planetary reference frames is set by the RNP model. Components, such as rotation-only, can be accessed, but the primary output would be the orientation of the planet's *pfix* reference frame with respect to its parent:

```
dr_group.add_variable( "earth.planet.pfix.state.rot.T_parent_this["+str(ii)+"]["+str(jj)+"]")
```

Note – the parent of a *Planet* object's *pfix* reference frame is always the same *Planet* objects's *inertial* reference frame.

Other useful logging values are described in section 4.1 of the RNP documentation.

C.5.2.d Model Dependencies

The RNP model requires three clocks from the Time Model: Terrestrial Time (TT), Greenwich Mean Sidereal Time (GMST) and Universal Time (UT1). It must also initialize with the Dynamics Manager.

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

Exercise 12. Creating a Non-spherical-Earth Orbiter

C.5.3 Mars-RNP

Follows a very similar pattern to Earth-RNP but with fewer components.

C.5.3.a S_define Setup

The Mars-RNP model typically resides in the Mars simulation-object.

RNP Declarations

With no nutation and polar motion component, there is only one class (RNPMars) to instantiate, along with its default-data class (RNPMars_rnp_mars_default_data).

```
verif/SIM_mars/SIM_mars_orientation/S_define
   284    jeod::RNPMars       rnp;
...
   292    jeod::RNPMars_rnp_mars_default_data    rnp_mars_default_data;
```

RNP initialization

```
   308      ("default_data") rnp_mars_default_data.initialize ( &rnp );
...
   316      P_ENV  ("initialization") rnp.initialize(
   317          internal_dynamics->manager );
   318      P_ENV  ("initialization") rnp.update_rnp(
   319          internal_jeod_time->tt );
```

RNP run-time calls

```
324      (LOGRATE, "environment") rnp.update_rnp(
325          internal_jeod_time->tt );
```

C.5.3.b Model Dependencies

The RNP model requires time input from the Time Model: Terrestrial Time (TT) is the most convenient form; there is some controversy over whether TT or Barycentric Dynamical Time (TDB) most closely matches with the old Ephemeris time (ET); some users may prefer to use TDB, although the difference is minimal.

C.5.4 Moon-RNP / Orientation

The lunar orientation is managed by the De4xx Ephemeris model; there is no specific RNP model for the moon. When the De4xx model updates, if the Moon's pfix frame is present the De4xx updates the lunar attitude. The method *propagate_lunar_rnp* may be called to update the lunar orientation if it is needed at a higher frequency than the regular De4xx model update is called (e.g. if it is needed at the derivative rate for computing non-spherical lunar gravity).

```
P_ENV ("derivative") de4xx.propagate_lunar_rnp ( );
```

However, recognize that the moon rotates very slowly ($O(10^{-4} deg/s)$) so this is rarely necessary. In cases where a lunar-vehicle is integrated in the Earth-inertial frame, the lunar position is usually more significant than the lunar orientation. Even in cases where a lunar-vehicle is integrated in Moon-inertial, the Earth-position is often more important than the lunar orientation. In these cases, the preferable strategy would be to run the entire Ephemerides model at the derivative rate. This can be accomplished with the flag setting

```
dynamics.dyn_manager.deriv_ephem_update = True
```

C.6 Ephemerides - Adding to the Environment Object

Document reference: *models/environment/ephemerides/docs/ephemerides.pdf*

C.6.1 Introduction

The Ephemerides model provides data-driven position information for each of the major planets in the Solar System. To use this model, the simulation must be running in the ephemeris-driven *Ephemerides* mode (see section B.3.3.a). Indeed, without this model, the simulation should not be running in *Ephemerides* mode, the model and the mode are inter-dependent.

Planetary state data is all referenced to the parent reference-frame of the Planet object's *inertial* reference frame (see chapter D for information on the reference-frame tree):

- For Earth and Moon, the parent reference-frame is the Earth-Moon Barycenter inertial frame.
- For other major planets, and the Earth-Moon Barycenter, the parent reference-frame is the Solar-System Barycenter inertial frame.

There are two implementations of the Ephemerides Model available: DE405, and DE421. Both are from the Jet Propulsion Laboratory. The default behavior is for DE405, but changing to DE421 is very easy (see section 4.2

in the Ephemerides Model document for details on this, or see the S_overrides.mk file below for a general syntax that makes the changeover easy).

Both models provide data for all of the planets in the Solar System, as well as other interesting points. JEOD identifies which of those objects are useful by looking for subscriptions to specific reference frames (see chapter D for information on reference frames). If a planet's (see section C.3) reference frames are not subscribed, it is not currently needed in the simulation and its state will not be computed.

In order to make the Ephemerides model work, the binary data files needed by the model must be built; this is typically done as part of the simulation compilation. This is the only instance in JEOD where we rely on external data manipulation during compilation, and it needs special treatment.

C.6.1.a S_overrides.mk File

Trick provides a mechanism in the S_overrides.mk file which can be augmented to produce the binary data files. Instructions are found in section 4.2 of the Ephemerides Model document on how to do this.

```
S_overrides.mk
#================================================================================
# Build simworld ephem files
#================================================================================
all: build_ephem_files

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

With this configuration, changing *EPHEM_MODEL* to 421 will switch the model to *DE421*.

C.6.2 S_define Setup

When initializing the Ephemerides Model, we need access to both the Time Manager and Dynamics Manager, so the first things to add are the appropriate references and arguments to the simulation-object class constructor.

```
 17 class EnvSimObject: public Trick::SimObject {
...
 26    EnvSimObject( jeod::TimeManager & time_manager_in,
 28                  jeod::DynManager & dyn_manager_in)
 29    :
 31       EnvironmentBasicSimObject (dyn_manager_in),
 30       time_manager(time_manager_in)
 32    {
...
 20    jeod::TimeManager & time_manager;
 17    jeod::DynManager & dyn_manager;
...
 2 EnvironmentBasicSimObject env(dynamics.dyn_manager);
```

C.6.2.a Model Declarations

We declare an instance of the Ephemerides Model

```
lib/jeod/JEOD_S_modules/Base/environment.sm
 24    De4xxEphemeris de4xx;
```

C.6.2.b Model Initializations

We have one model initialization job:

```
lib/jeod/JEOD_S_modules/Base/environment.sm
...
 34       P_EPH  ("initialization") de4xx.initialize_model ( time_manager,
 35                                                          dyn_manager);
```

There are no run-time function calls needed to update the model; the Dynamics Manager will perform that task.

C.6.3 Setting the Input File Values

There are a limited number of modifications that may be made to the Ephemerides Model, including de-activating the model, and moving the update rate from its “scheduled” value to the derivative rate. See the Ephemerides Model document for information on these.

C.6.4 Logging Data

The Ephemerides Model is used to populate the state of the planetary reference frames. It is not intended that users should ever need to look at any data directly from the Ephemerides Model. Instead, we can look at the populated states, for example the position of the *planet*'s inertial reference frame as found in the simulation object *sun*, or the velocity of the *planet*'s inertial reference frame, as found in the simulation object *earth*.

```
models/environment/ephemerides/de4xx_ephem/verif/SIM_ephem_verif/Log_data/log_planetary_eph
em_rec.py
24     dr_group.add_variable("sun.planet.inertial.state.trans.position[" + str(ii) + "]" )
...
30     dr_group.add_variable("earth.planet.inertial.state.trans.velocity[" + str(ii)+ "]" )
```

C.6.5 Model dependencies

The Ephemerides Model does require input from the Terrestrial Time (TT) clock. Notice that the *initialize_model* function call takes the Time Manager as an argument; this provides the tie to the TT clock. The TT clock should be defined in the time simulation-object and registered with the Time Manager (see section B.2 for details on these steps)

C.6.6 Multiple Gravity Fields

With the Ephemerides Model, we can now have multiple correctly positioned planets in a single simulation. Each of those planets has a gravity field; depending on the situation, it may be desirable to include the effects of multiple gravitational fields. The primary gravitational field will be from the planet that defines the integration frame. Other planets will either provide accelerations derived from Newtonian gravity, or 3rd-body perturbing effects. This is explored in more detail in section D.7.

Exercise 13. Multiple Gravitational Fields

Exercise 14. Lunar Orbiting Vehicles

C.7 Gravity Gradient Torque – Adding to the Vehicles

Document reference:
models/interactions/gravity_torque/docs/gravity_torque.pdf

C.7.1 Introduction

The gradient of the gravitational field is already computed in generating the gravitational acceleration of the vehicle. By applying that gradient to the vehicle inertia tensor, we derive the torque effect that an asymmetric vehicle is subject to, resulting from one part of the vehicle receiving a stronger gravitational acceleration than another. This is known as the gravity-gradient torque.

Typically, the gradient becomes negligible at large distance, and although it is technically possible to compute the gradient torque for distant gravitational bodies (such as Sun when in Earth orbit), it is a futile effort to do so.

The torque is computed in the vehicle body frame and transformed to the vehicle structural frame for output. This output is collected in the same way as any other external force or torque (excluding gravitational acceleration), with one important caveat:

THE GRAVITY GRADIENT TORQUE IS A NO-TRANSMIT-TORQUE. When we have two (or more) vehicles connected together, we only propagate (integrate in time) the position of one (the root body), and derive the positions of the other(s) (the child(ren)) locally as necessary. If a child body is subject to some force or torque, we typically transmit that up to the parent for inclusion in the overall propagation. That is not the case with gravity gradient. Because the parent body's inertia tensor already includes the inertia of all of its children, transmitting the torque of a child up to the parent would result in double-counting the effect. It is necessary to compute the gravity gradient for the one composite body of parent and child together (rather than computing the torque independently and adding, as we would if we had a thruster on each body) because the torque depends on the inertia tensor, which is affected by not only the inertia of the two bodies, but also their relative positions.

C.7.2 S_define Setup

The gravity torque model is typically added to the vehicle simulation-object, with a separate implementation for each vehicle. It is an easy model to add, with one declaration, one initialization, one run-time call and one

collect statement. The run-time call is typically performed at the derivative rate, and given a low priority (P_BODY) since nothing depends on it.

```
(based on interactions/gravity_torque/verif/SIM_torque_compare_simple/S_define)
504 ##include "interactions/gravity_torque/include/gravity_torque.hh"
...
509 class Sv_dynSimObject: public Trick::SimObject {
...
512     jeod::DynBody       body;
...
530     jeod::GravityTorque grav_torque;
...
580       ("initialization") grav_torque.initialize( body );
603       P_BODY   ("derivative") grav_torque.update();
...
611 Sv_dynSimObject sv_dyn ( ... )
...
624 vcollect vehicle.body.collect.collect_no_xmit_torq jeod::CollectTorque::create {
625    vehicle.grav_torq.torque};
```

Notice that the *initialize* method takes the DynBody instance as an argument.

C.7.3 Setting the Input File Values

In the input file, the gravity gradient torque can be turned on or off, and the resolution of the model used to compute the torque can be set. The degree and order for the gradient computation must be no larger than that set for the linear acceleration; consequently, if the gravitational field is set to spherical (degree and order are zero), the gradient is also restricted to that of a spherical model. If the degree and order are not specified, they default to 0 (spherical).

```
example
vehicle.earth_grav_controls.gradient = True
vehicle.earth_grav_controls.gradient_degree = 5
vehicle.earth_grav_controls.gradient_order = 5
```

C.7.4 Logging Data

The output of the model is stored in the *torque* array:

```
example
dr_group.add_variable( "vehicle.grav_torque.torque["+str(ii)+"]")
```

Exercise 15. Gravity Gradient

C.8 Solid Body Tides

Document reference: models/environment/gravity/docs/gravity.pdf

C.8.1 Introduction

Consider a high-precision simulation of a vehicle in Earth orbit, utilizing a high-resolution gravitational field. The coefficients used to derive the gravity gradient (for acceleration or torque) are mean values, the actual field fluctuates over time because Earth's moon distorts Earth as it moves in its own orbit. The Solid Body Tides Model provides temporal variations in the gravity coefficients to account for 3rd-body (e.g. Moon) effects on the gravitational body itself. This is quite distinct from 3rd body effects on the vehicle.

The tidal effect is calculated (not based on a data table). Consequently, it is necessary to model the planets that are going to be providing the tidal effect. For example, the dominant effect on Earth is from Moon, so Moon must be modeled as a planetary object. Sun should also be included, since its effect is also significant.

C.8.2 S_define Setup

As with the Gravity Manager, this is typically included in the planet simulation-object with vehicle specific controls added to the vehicle simulation-object. The planetary component can be populated with the default data for Earth, provided in the gravity model.

C.8.2.a Declarations

First, in the planet object:

```
environment/gravity/verif/SIM_tide_verif/S_define
 52 ##include "environment/gravity/include/spherical_harmonics_solid_body_tides.hh"
 53 ##include "environment/gravity/include/spherical_harmonics_solid_body_tides_init.hh"
...
 56 ##include "environment/gravity/data/include/earth_solid_tides.hh"
...
 62     jeod::SphericalHarmonicsSolidBodyTides sb_tide;
 63     jeod::SphericalHarmonicsSolidBodyTidesInit sbtide_init;
...
 66     jeod::SphericalHarmonicsSolidBodyTidesInit_earth_solid_tides_default_data
           sbtide_init_default_data;
```

and in the vehicle object:

```
environment/gravity/verif/SIM_tide_verif/S_define
123 ##include "environment/gravity/include/spherical_harmonics_delta_controls.hh"
...
144     jeod::SphericalHarmonicsDeltaControls sbtide_ctrl;
```

Note – the default data provided for Earth assumes that Earth's body-tides are governed by Sun and Moon. Both Sun and Moon must therefore be modeled in the simulation to be able to use this default data file.

C.8.2.b Initialization

In the planet object, the default data instance is used to populate the initialization class. Then the initialization class is passed into the *add_deltacoeff* method, (along with the Dynamics Manager, and the instance of the Solid Body Tides Model)

```
(based on environment/gravity/verif/SIM_tide_verif/S_define)
 85       ("default_data") sbtide_init_default_data.initialize ( &sbtide_init );
...
 88       P_EPH  ("initialization") gravity_source.add_deltacoeff ( sbtide_init,
 89                                                                 dyn_manager,
 90                                                                 sb_tide );
```

(where dynamics_manager is a reference to the instance of DynManager from the dynamics simulation-object)

With this call, the Dynamics Manager has knowledge of the adjustments to the gravity coefficients that it will need when it calculates the gravity gradient.

Note the priority on this call. The only requirement here is that the *planet.register_model* call precede the *gravity_source.add_deltacoeff* call (the former registers the planet with the Dynamics Manager, and the latter uses that index to find the planet). This call could also be P_ENV priority with appropriate ordering within the S_define if the call sequencing, but we have chosen to use P_EPH rather than relying on the order in which the calls appear in the S_define to avoid the potential for editing errors.

C.8.2.c Runtime

There is no run-time call necessary because the Dynamics Manager will handle everything after initialization is complete.

C.8.3 Setting the Input File Values

The vehicle delta-controls are set in much the same way as the vehicle gravity controls. They need associating with the solid-body tides instance and with the gravity source:

```
based on environment/gravity/verif/SIM_tide_verif/SET_test/RUN_01/input.py
 94 sv_dyn.sbtide_ctrl.grav_effect = earth.sb_tide
 95 sv_dyn.sbtide_ctrl.grav_source   = earth.gravity_source
```

the two controls available are to turn the effect on or off with the *active* flag, and to specify the resolution.

```
 92 sv_dyn.sbtide_ctrl.active = True
 93 sv_dyn.sbtide_ctrl.first_order_only = True
```

As with the gravity controls, the delta-control must be added to the simulation. This is intrinsically tied to the adding of the gravity controls to the vehicle; without gravity controls there can be no delta-controls. Hence, this is a two-part process. The delta-controls get added to the gravity controls, and the gravity controls get added to the vehicle. The latter process should already be in place from establishing the regular gravity controls, it does not need to be repeated.

```
97 sv_dyn.earth_grav_ctrl.add_deltacontrol(sv_dyn.sbtide_ctrl)
98 sv_dyn.body.grav_interaction.add_control(sv_dyn.earth_grav_ctrl)
```

Exercise 16. Solid-Body Tides

C.9 Omitted Environment Models

C.9.1 Atmosphere

Although establishing an atmosphere in the simulation logically belongs in the Environment chapter, the implementation of the atmosphere requires knowledge of Derived States, which we have yet to cover. Furthermore, the usage of the atmosphere – and the ability to test the configuration – has to wait until we cover aerodynamic drag, which we also have yet to cover.

Consequently, the details on adding an atmosphere have been moved to section G.2.

C.9.2 Earth Lighting

Earth Lighting is covered in section I.1.

Chapter D Behind the Scenes - Reference Frames and Trees

The tree structures in JEOD have caused users headaches in the past – they are not a trivial concept and need some explanation. Consequently, we have chosen to divide this topic into two chapters. This chapter provides conceptual information on the management of reference frames, the reference frame tree, and the mass tree. Application and instructions for integrating these models into a simulation is found in Chapter E. If you are comfortable with tree structures, much of this chapter can be skimmed.

D.1 Introduction to Reference Frames

Document reference: models/utils/ref_frames/docs/ref_frames.pdf

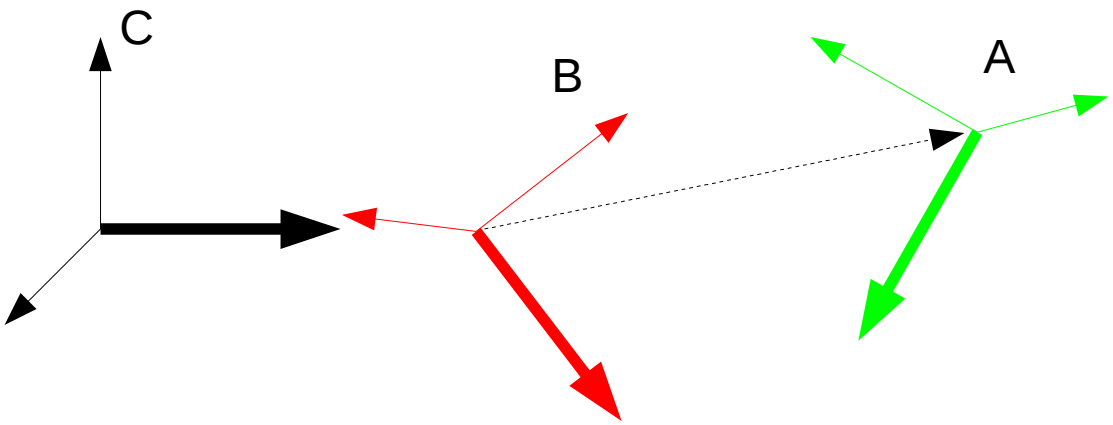
D.1.1 What Reference Frames Do

Reference frames provide a means by which we can:

- Represent the **state** of an object in a meaningful way. A state comprises:
  - position
  - orientation
  - velocity
  - angular velocity

  all of which are expressed relative to some other frame.
- Provide a safe setting for integration.
- Make mathematical calculations easier, such as by utilizing principal axes in rotation calcualtions.

The relative state of one frame (A) relative to another (B) is often expressed as a set of 3-vectors, using axes that are defined by some reference frame. Sometimes, the preferred choice of frame in which to express the value is the frame whose state is being measured (A), sometimes it is the relative frame (B), and sometimes it is a completely different frame.



In the following discussion of reference frames, you will see statements such as: the *state* of frame *A* relative to frame *B* expressed in frame *C*. By default, unless specified otherwise:

- the position and velocity of frame *A* relative to frame *B* are typically expressed in the relative frame (*B*)
- the angular velocity is typically expressed in frame *A*
- the relative orientation may be expressed in multiple formats, including transformation matrix, quaternion, Euler angles, Eigen rotations.

D.1.2 Reference Frames and Objects

A physical object, such as a vehicle, has one or more reference frames associated with it. Those frames have states. Objects do not have states, objects are just physical entities; it is their frames that have the states. When we loosely refer to the *state* of an object we are most likely meaning the state of the object's body-reference-frame.

Most objects in JEOD fall into one of three categories:

1. *Planet. Planet* objects have two fundamental reference frames. Both have origins at the center of the planet. One (*planet-fixed*) has axes that rotate with the planet, and the other (*inertial*) has axes that are always pointing in the same direction – the same direction is used for all planets.

   Note – while a planet's *inertial* frame may be accelerating, it is referred to as an inertial frame because its axes are aligned with an external frame that, at least relative to the background stars, is not accelerating. Two inertial frames, such as Earth-inertial and Solar-System-Barycenter inertial may have a small relative acceleration, but their axes will always be aligned.

2. *MassBody.* A *MassBody* is a representation of a (non-planetary) massive object. It has information about mass and mass distribution, but has no reference frames. Consequently, the state of a MassBody is a meaningless concept.

3. *DynBody.* A *DynBody* is an extension of a *MassBody,* with the addition of reference frames (which have states). There are at least three reference frames associated with a *DynBody:*

   - Core-body – Origin is at the center of mass of the body; axis-orientation defines the body-axes.
   - Composite-body – Origin is at the center of mass of the composite body – that is, the body, and everything attached *to* it. Axis-orientation is the same as for the core-body.
   - Structure – Origin is at the structural origin, a point often defined relative to some structural feature of the vehicle. Axis-orientation is also often defined relative to structural features.

In defining a *DynBody*, it is possible to add additional reference frames beyond these three. For example, suppose a vehicle has an accelerometer; in order to know the state at the location of the accelerometer, there must be another reference frame anchored there. The three reference frames outlined above are all instances of the *BodyRefFrame* class and we can add as many of these to a DynBody as we choose. More information on this later in section E.2.2.

D.1.3 Reference Frame Examples

An LVLH frame has a state relative to some planet inertial frame. An LVLH frame is usually attached to some vehicle, so the state of the LVLH frame is often the same as the state of the vehicle's body frame.

A vehicle's (a *DynBody*) structure frame has a state relative to that LVLH frame.

The vehicle's structure frame also has a state relative to the planet inertial frame.

The position of the vehicle's structure frame relative to the LVLH frame could be expressed in the LVLH axes, or the planet inertial axes, or the vehicle's structural axes.

D.2 Reference Frames in JEOD

D.2.1 Subscribing Reference Frames.

The Dynamics Manager handles the calculation of all reference-frame states behind the scenes, but not all reference frames are required in all parts of a simulation. Calculating unnecessary states is wasteful, so the Dynamics Manager has the functionality to determine whether or not to calculate the state of all reference frames registered with it. Basically, when a model needs a particular reference frame, it subscribes to it. When it no longer needs it, it unsubscribes from it. The Dynamics Manager keeps track of how many subscribers each frame has, and calculates the state only of those frames with at least one subscriber.

D.2.2 Handling Reference Frames

In JEOD, all reference frames are named and registered with the Dynamics Manager. For the most part, frame names are automatically generated following a straightforward convention; there is no requirement that the user name all of the reference frames, or even be aware of all of them in many cases.

JEOD stores the state of a frame with respect to its parent in the variables:

- <*frame_name*>.*state.trans.position*, 3-vector expressed in the parent frame
- <*frame_name*>.*state.trans.velocity*, 3-vector expressed in the parent frame
- <*frame_name*>.*state.rot.T_parent_this*, transformation matrix from the parent to the <*frame_name*> frame. Transformation matrices are not expressed in any frame, they are frame neutral.
- <*frame_name*>.*state.rot.ang_vel_this*, 3-vector expressed in the <*frame_name*> frame.

For these variables, <*frame_name*> is an instance of the *RefFrame* class (or derivative thereof), and *state* is the instance of the *RefFrameState* class, included in all *RefFrame* instances. Both of these classes are used

extensively in JEOD. The Derived State Model (see chapter H) also provides additional formats for expressing relative states.

D.2.2.a Frame Naming Convention

Frames are named according to one of two paradigms:

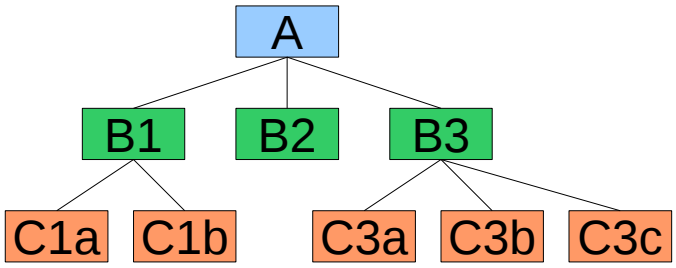
- For planetary-based or DynBody-based frames, use <object-name>.<frame-type>. For example:
  - *vehicle.composite_body*
  - *Earth.inertial*
- For reference frames associated with Derived States, use <subject-name>.<reference-name>.<frame-type>. For example,
  - *vehicle.Earth.lvlh* for the LVLH frame associated with vehicle referenced to Earth.

D.3 The Reference Frame Tree

NOTE – It is usually not necessary to know the details of the tree structure, but it is often helpful to maintain at least a crude overview of how the frames interrelate.

D.3.1 The Tree Concept

The tree is a way to show the dependence of one instance on another. With one and only one exception, all reference frames in the simulation have one and only one *parent*, relative to which the state of the frame is measured. Since there is only one frame in the simulation with no parent, all of the frames must be connected in some sense, in a tree structure. The diagram below illustrates this structure, showing that, for example, the state of frame B1 is known relative to frame A, and the state of frame C3a is known relative to frame B3.



Since each frame has only one parent (its state is expressed relative to only one other frame), there can be no multiply-connected elements in the tree. Also, as the tree is constructed, verification is made that there are no isolated groups of frames with circular references. With knowledge of the tree structure, and knowledge of the state of every frame relative to its parent, it is possible to trace through the tree to obtain, for example, the state of frame *C1a* relative to *C3c*.

D.3.2 The Tree Root

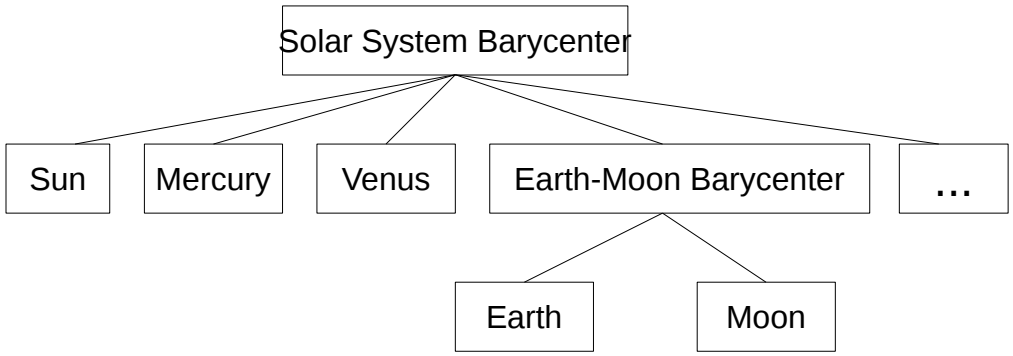
The one exception to the rule that all frames have one and only one parent is for the ***root frame***. The root frame has no parent; it is the frame that **defines** zero state (translational and rotational, position and velocity) in the simulation.

When running in *EmptySpace* mode, the root frame is the inertial frame associated with the somewhat arbitrary, user-specified, *central_point_name* value (i.e. *<central_point_name>.inertial*). The orientation of this frame has no particular meaning, although other frames may be oriented with respect to it.

When in *SinglePlanet* mode, the root frame is the inertial frame of the Planet object (as with *EmptySpace* mode, this is specified through the *central_point_name* value). Although the orientation of the inertial frame in a universe with just one planet would also be entirely arbitrary, we need some datum orientation for interfacing between models. Consider the RNP model, which calculates the planetary orientation as a function of time. The algorithm used by that model must be universally applicable (to all simulations), hence the reference frame it uses as a datum must be defined external to the simulation. Consequently, in order to have an a priori

description of the planetary orientation with respect to the planetary inertial frame, the planetary inertial frame must be likewise externally defined. To meet this requirement, JEOD aligns all of the planetary inertial reference frames with the International Celestial Reference Frame (ICRF).

When running in *Ephemerides* mode, the root of the tree is generated automatically from the following diagram such that the tree is as small as possible. The planet names refer to their inertial reference frame.



For example, for an Earth-only simulation, the root of the tree will be Earth.inertial.

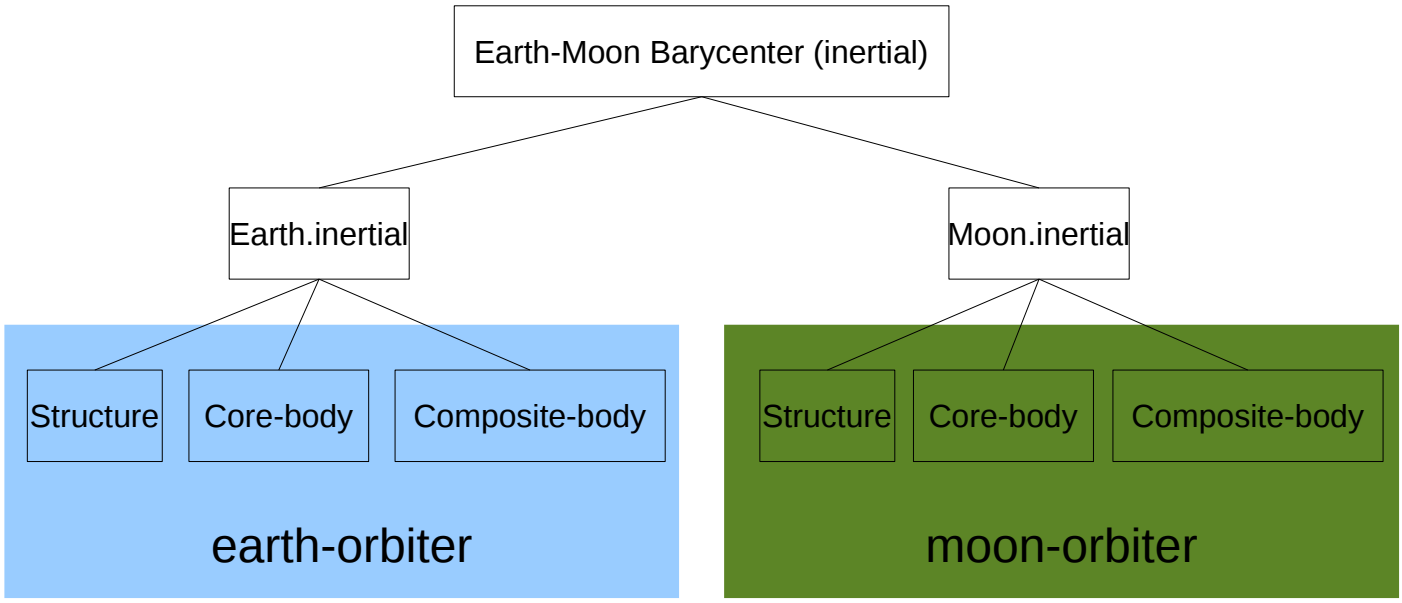
For an Earth-Moon simulation, the root of the tree will be Earth-Moon Barycenter with Earth and Moon as children.

For an Earth-Sun simulation, the root of the tree will be Solar System Barycenter, with Sun and Earth-Moon Barycenter as children. In this case, Earth-Moon Barycenter has a single child, Earth.

D.3.3 Extending the Tree

The reference frames associated with *DynBody* objects are added to the tree by the Dynamics Manager. Each *DynBody* has an assigned integration frame; integration frames are inertial frames that already exist in the ephemerides-defined tree (illustrated above). The Dynamics Manager adds the three *DynBody* frames as children to that integration frame.

For example, consider a two vehicle simulation with one vehicle integrated in *Earth.inertial*, one integrated in *Moon.inertial*, and no other planetary objects involved. Because Earth and Moon are the only planets, the root of the tree will be Earth-Moon Barycenter, and the vehicle reference frames will be attached to their respective inertial frames.



In this example, each vehicle was attached to a different integration frame, but that is not a requirement. Each inertial frame can support any number of vehicles, with all *DynBody* frames added at the same level, as siblings.

The tree can be further extended manually; this is discussed in chapter E.

D.4 The Mass Tree

Document reference: models/dynamics/mass/docs/mass.pdf

D.4.1 The Mass Tree and the Reference-Frame Tree

There are two trees in use in JEOD, and they are often confused because they represent similar things and have similarly named entities. Recall that a vehicle does not have a state, it has reference frames that have states. The linkage between reference frames is represented in the reference frame tree. A vehicle also has mass properties (see D.4.3), and these, too, may have a hierarchy. Mass property dependencies are represented in the Mass Tree. The two trees are completely independent.

To add to the confusion, some of the naming conventions are similar. On the reference frame side, we have *structure*, *core-body*, and *composite-body* frames. On the mass properties side, we have *structure-points*, *core-properties*, and *composite-properties*. The two sets of data are closely related – hence the similar names – but are conceptually independent. While this like-naming convention can be confusing to someone just starting with JEOD, you will soon see the similarities between these two trees, and the two sets of like-named values, and realize why the naming is so similar.

Until the distinction is soundly understood, be very careful when considering trees and elements of the trees to ensure that there is no migration of concepts from one to the other.

D.4.2 Combined *MassBody* elements

MassBody objects can be attached together to make larger bodies. Whenever a vehicle can operate as two components, or has two components whose behavior differ, it should be created as two MassBody objects and attached.

Examples:

- Single vehicle becoming two. A multi-stage vehicle would have a first-stage and a second-stage that are initially attached but detached at a later time.
- Separate vehicles becoming one. A docking scenario would start with two vehicles that attach at some point and become one.
- Components that will always be attached. Local mass depletion, such as with fuel tanks, can be modeled by representing the vehicle as a shell with attached fuel tank objects. Modeling the inertia of the fuel tanks during depletion may be easier than modeling the inertia of the whole vehicle, but if the tanks are attached to the vehicle, the propagation of a changing tank-inertia gets propagated to the whole vehicle automatically.

D.4.3 Mass Properties

Mass properties provide information on the mass, inertia tensor, and position of the center of mass. Each set of properties also defines a partial reference frame, providing position and orientation only. All *MassBody* objects (whether they be in *MassBody* form or the extended *DynBody* form) have two sets of mass properties.

D.4.3.a core_properties

Core properties are those of the elemental *MassBody* – that is, the *MassBody* with nothing attached to it.

D.4.3.b composite_properties

Composite properties are those of the elemental *MassBody* and everything attached *to* it. Note that attachments have a relational hierarchy, there is a child and a parent. The composite properties of the parent include both vehicles because the child is attached *to* the parent; the composite properties of the child include the child only because the parent is NOT attached *to* the child.

D.4.4 Mass Points

Where there are additional points of interest on a vehicle (e.g. an antenna, a docking port), we can add additional *MassPoint* objects to represent those places. There are two classes of mass points, the base-class *MassBasicPoint*, and the *MassPoint* class that inherits from *MassBasicPoint.*

Note – *MassProperties* also inherits from *MassBasicPoint*, adding mass and inertia information. Hence, *MassBasicPoint* and *MassPoint* instances do not have mass or inertia information.

D.4.4.a structure_point

Both *core_properties* and *composite_properties* have a position and orientation; those have to measured with respect to something; that something is called the *structure_point*, a *MassBasicPoint* instance. Each *MassBody* has a *structure_point.*

Recall that in a *DynBody*, the origin of the *core_body* reference frame is at the center of mass of the core-body. The position associated with *core_properties* is the position of that core-body center of mass relative to the *structure_point*, whereas the position associated with the *core_body* reference frame is measured with respect to the integration frame. Recall that the *DynBody* also has a reference frame called *structural.* By configuring these points such that the *structure_point* matches the position and orientation of this *structural* reference frame, and the orientation of the *core_properties* matches that of the *core_body* reference frame, we have the relation between the two sets of data. Then, by defining the position and orientation of the *core_properties* (with respect to the *structure_point*) we also define the relative position and orientation of the *core_body* reference frame with respect to the *structural* reference frame. An identical argument can be made for *composite_properties* and the *composite_body* frame.

Note, however, that there is a hierarchy in the mass properties that does not exist in the reference frames. Recall that the *core_body* reference frame, *composite_body* reference frame, and the *structural* reference frame are siblings in the reference frame tree; their states are both measured with respect to another frame (the inertial integration frame). Conversely, the "state" of *core_properties* and *composite_properties* are measured with respect to the *structure_point*, so *core_properties* and *composite_properties* are conceptually like children of the *structure_point* on the reference frame tree *(*conceptually only, remember *core_properties*, *composite_properties* and *structure_point* are NOT reference frames, so do NOT get placed on the reference frame tree).

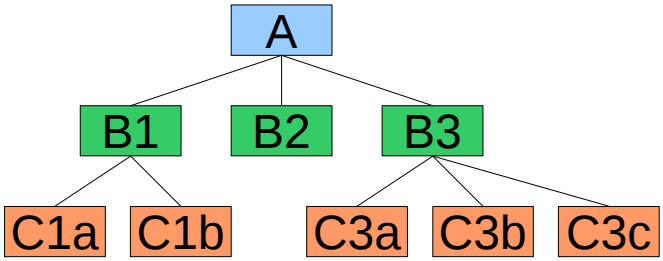
D.4.4.b Other Mass Points

Like *MassProperties*, a *MassPoint* has a position, an orientation, no reference-frame, and no state. The position and orientation are evaluated with respect to the parent *MassBasicPoint*, just as the position and orientation of *core_properties* is evaluated with respect to its parent, the *structure_point.*

MassPoint objects are typically declared with respect to the *structure_point* of a *MassBody.* Like the *core_properties* and *composite_properties*, they are conceptually like children of the *structure_point.*

D.4.5 The Mass Tree Concept

Unlike the reference frame tree, there may be multiple mass trees in one simulation. Only objects that are physically attached to one another share the same mass tree. Like the reference frame tree, each mass tree has a root object, and each non-root tree-element has one and only one parent with no circular references allowed. However, while there may be only one reference-frame tree in a simulation, that same simulation may have multiple mass trees. This tree shows the hierarchy of which objects are attached to which other objects. For example, in the diagram below, mass B1 is attached to mass A; mass C3a is attached to mass B3.



The composite-properties of a mass include the mass itself, and everything that is attached to it in this hierarchy.

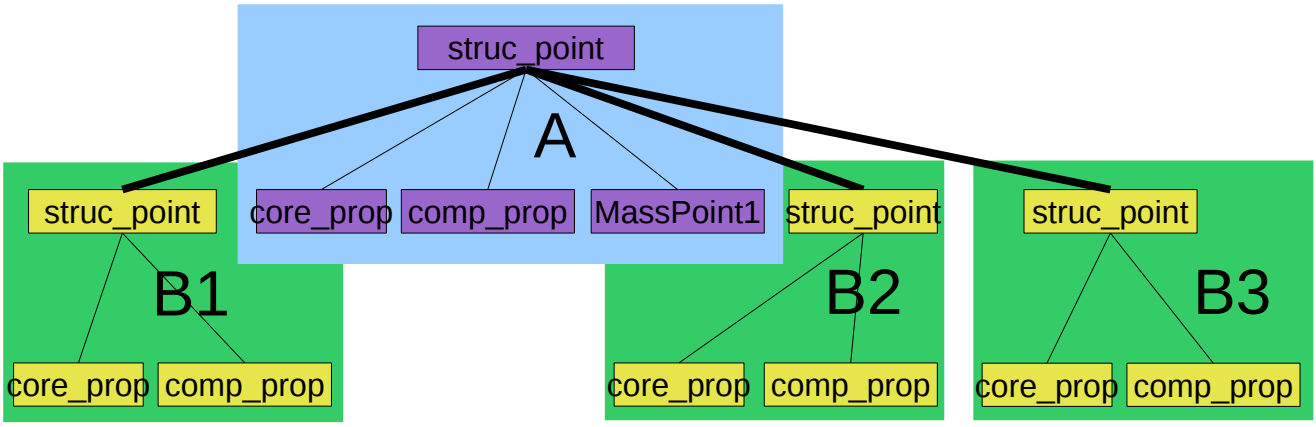
In this example, the composite-properties of object A include the composite-properties of B1, B2 and B3. Or, put another way, the composite-properties of object A include the core-properties of B1, B2, B3, C1a, C1b, C3a, C3b, and C3c.

D.4.5.a Mass Tree, MassBody, MassPoint, MassProperties, and MassBasicPoint

From a conceptual viewpoint, it is most useful to think of the mass tree as a collection of MassBody objects, but that is not technically correct.

The mass tree is actually a tree of *MassBasicPoint* objects (thereby representing *MassPoint* and *MassProperties* objects since they inherit from *MassBasicPoint*). A *MassBody* has two *MassProperties* objects *(core_properties* and *composite_properties)*, one *MassBasicPoint* (*structure_point*), and a list of *MassPoint* objects. The *MassProperties* and *MassPoints* are made children of the *structure_point*. When one *MassBody* attaches to another, its *structure_point* is made a child of the parent's *structure_point*, thereby becoming a sibling of the parent's *MassProperties* and *MassPoint* instances.

The first two rows in the mass tree illustrated above are shown in more detail below, with the bold lines indicating the previous connections, and a *MassPoint* added onto *MassBody* A:



Behind the scenes, it gets even more complicated, but the vast majority of users do not need to consider these advanced concepts. A *MassBody* has additional *MassBasicPoint* instances, called *composite_wrt_pbdy*,

core_wrt_composite, and *composite_wrt_pstr*. These are all used for maintenance of core and composite properties, and they have the following attachments:

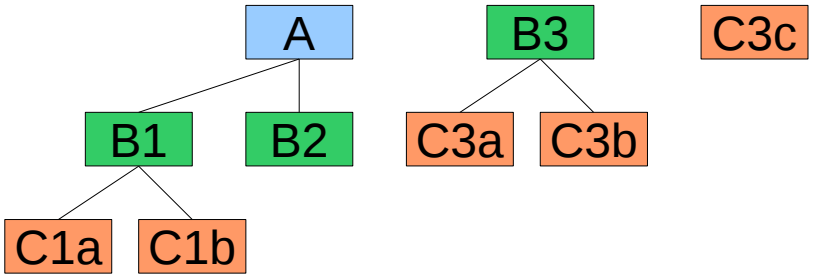
- *core_wrt_composite* attaches to the *composite_wrt_pbdy* of the same *MassBody*.
- *composite_wrt_pbdy* attaches to the *composiste_wrt_pbdy* of the parent *MassBody*
- *composite_wrt_pstr* attaches to the *structure_point* of the parent *MassBody*

Putting these advanced details aside, we will now continue with the simpler – albeit only conceptual – mass tree that represents the masses within the simulation.

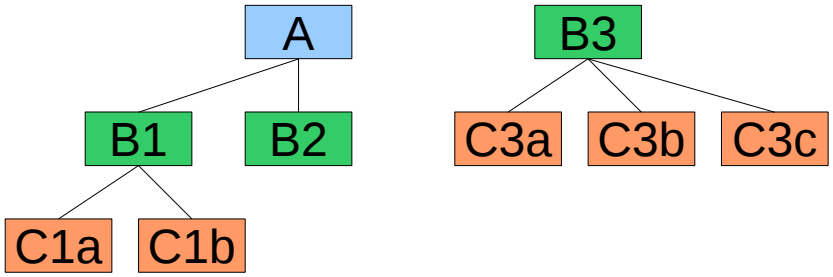
D.4.5.b Building Mass Trees

A Mass tree may be built by attaching two *MassBody* objects. This process is often performed with a body-action, which we will investigate in section F.6. It may also be performed directly without the use of body-actions; see section F.6.5 for description of this method.

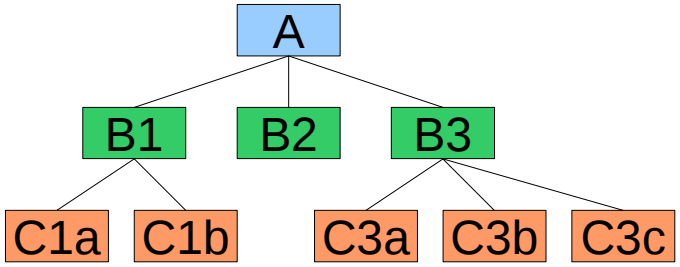
Mass Trees may be built either by adding one *MassBody* at a time, or by combining two existing trees. For example, consider the following scenario in which there are three vehicles, with root objects A, B3, and C3c.



The atomic tree C3c can be attached to another vehicle, say to the root body B3, leaving two trees:



Now if we attach the body B3 to the body A, the entire tree of which B3 is root will attach to the tree of which A is root, and we are back to full tree.

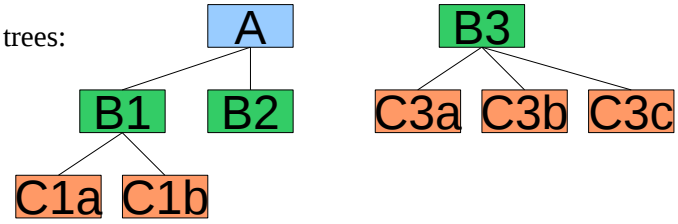


Exercise 17. Interpretation of Variables

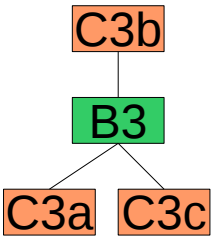
D.4.6 Abstract Nature of Mass Trees

It must be pointed out that the connections in a mass tree should not be assumed to represent real physical connections. The components of a mass tree must have a physical connection, but it is not necessary that the physical connection be represented by the parent-child connecting lines of the tree. Furthermore, the components of any mass sub-tree must be physically connected to each other, and one of those sub-tree components must be physically connected to the parent of the sub-tree.

Consider the previous example again with two trees:



Now, suppose we wanted to physically attach body C3b to A. We could, conceptually, “pick up” the tree at C3b to give:



and attach that to A. But this operation leads to significant complications. Recall the complexity of the real mass tree, coupled with the restriction (that we will soon discover) that *MassBody* objects that are *DynBody*

objects cannot be children of *MassBody* objects that are not *DynBody* objects. Realistically, the practice of restructuring the tree becomes formidable at best, and often impossible.

Instead, we simply attach the two trees as we did before, and rely upon the realization that the tree-connections do not always reflect physical-connections.

D.5 Reference Frames, DynBody Objects, and Integration

Document reference: models/utils/integration/docs/integration.pdf

There can be multiple *DynBody* instances in a simulation

- All *DynBody* objects have 3 reference frames, and all reference frames sit on the same reference frame tree.
- All *DynBody* objects containt *MassBody* objects
  - Attachments between *MassBody* objects show up on the mass trees
  - Each set of attached *MassBody* objects has its own tree
  - Each *MassBody* tree has a root
  - There is some number of *DynBody* objects that are a root of a mass tree, let us call these root-*DynBody* objects.
  - Note that hierarchy in a mass tree is not duplicated in the reference frame tree

During state integration, only those *DynBody* objects that are a root of a mass tree are integrated, and only one reference frame is integrated per root-*DynBody* (the integrated frame is the *composite_body* frame). Each integration is performed in some inertial integration frame, which is specified in the input file with the command:

```
vehicle.dyn_body.integ_frame_name = “Earth.inertial”
```

Note that this frame must exist (the planet must be defined and registered with the Dynamics Manager), and the frame must be inertial!

There may be more than one integration frame at any time in a simulation, but only one integration frame per root-*DynBody* (remember, there may be multiple root-*DynBody* objects).

The integration frame may be changed mid-simulation (see section I.4 for information on this). This may be desirable, for example, in an Earth-Moon transfer scenario:

- The vehicle starts near Earth and is integrated in *Earth.inertial*
- As the vehicle nears Moon, the integration frame can be switched to *Moon.inertial*.

The other reference frames (for the root *DynBody* and all *DynBody* objects attached to it within the mass tree) are populated based on the geometry information embedded within the *MassBody* objects that underlie the

DynBody objects. In the reference frame tree, all of these other reference frames are siblings to the frame that is being integrated.

This propagation of state through the tree leads to a very important consequence. Because *MassBody* objects do not have state, the propagation terminates when it reaches a *MassBody.*

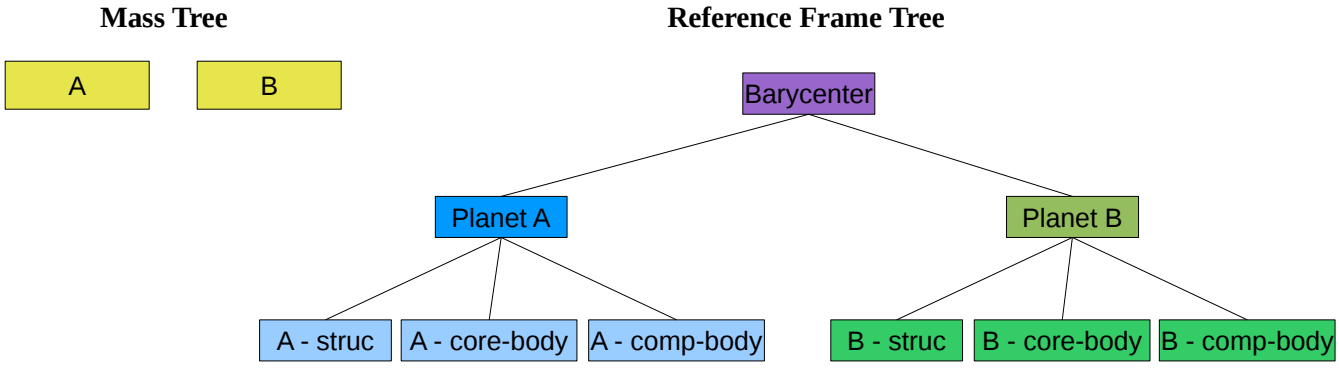
CONSEQUENTLY, *DynBody* OBJECTS CANNOT BE CHILDREN OF *MassBody* OBJECTS.

D.6 Reference Frame Manipulation on MassBody Attachments

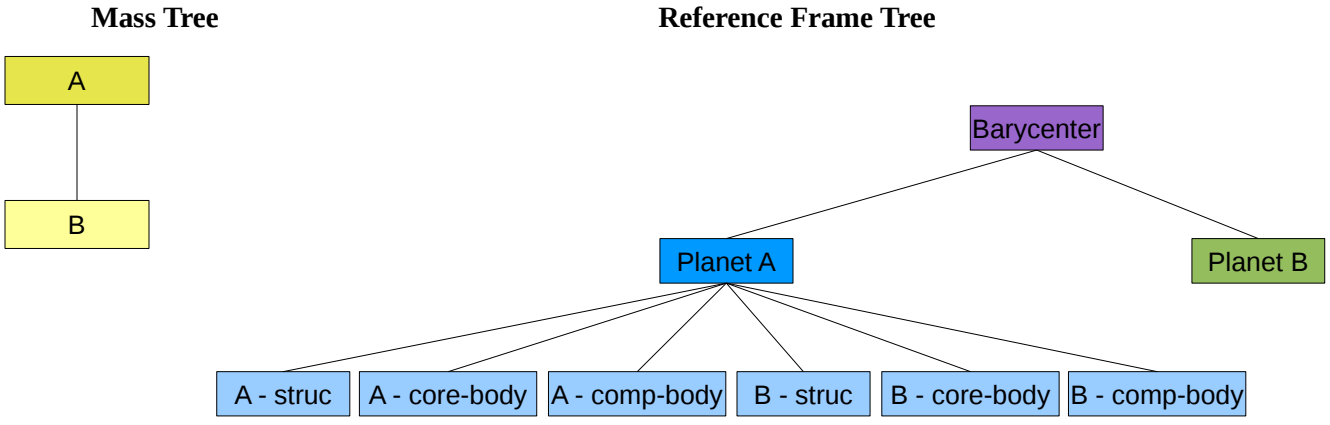
When two *DynBody* objects become attached, the child *DynBody* gets attached to the parent *DynBody* in the mass tree. The child *DynBody* is now no longer a root-*DynBody*, and so will not be integrated, and the child *DynBody* object's reference frames must now be made children of the parent *DynBody* object's integration frame.

In the case that the two *DynBody* objects initially had different integration frames, then in the reference frame tree, all of the frames associated with all of the *DynBody* objects in the child *DynBody* object's former mass tree must be be detached from its integration frame. They must all be added as children to the parent *DynBody* object's integration frame (and as siblings to the parent *DynBody* object's reference frames).

Example: Two independent vehicles in the mass tree, each integrated in its own integration frame.



When vehicle B is attached to A, both trees get rearranged:



Note - If the parent or child *DynBody* were not root-*DynBody*, consider their respective integration frame to be that of the root of their respective mass trees.

D.7 Gravity, Ephemerides, and the Reference Frame Tree (optional)

With a sound understanding of the reference-frame tree, we can now investigate the rules for gravitational computations.

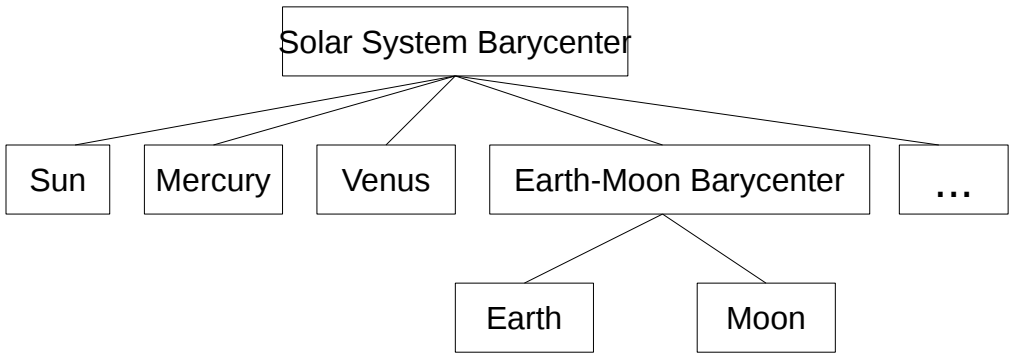
Consider the following hypothetical/illustrative example (don't actually try to make a simulation to do this):

Consider a vehicle being integrated in an inertial reference frame in proximity to some planet. For simplification, consider the vehicle and the frame to be at the same distance from the planet center, but separated tangentially by some distance. Both the vehicle and the integration reference frame are released from "rest" (relative to a planet) and allowed to free-fall in the presence of the planet's gravity (recall that when we talk about inertial frames, it is only meaning that the axes are aligned, not that the origin of the frame is non-accelerating). We could compute the gravitational acceleration of the vehicle relative to the planet, and express that in the integration frame ready for the integration of state, but that would be wrong.

Because the integration frame is also accelerating at the same rate as the vehicle, the "vertical" position of the vehicle in the integration frame should not be changing. In this case, it would be inappropriate to integrate the vehicle's gravitational acceleration due to the planet's gravity. But that is not to say that there is no relative acceleration. Because the frame and the vehicle are both accelerating radially, their tangential separation distance will be decreasing; there will be a relative acceleration, but it will be in the planet's horizontal direction. We should be applying to the vehicle the difference in the gravitational acceleration experienced by the vehicle and by the integration frame. This difference is known as a 3rd-body perturbation.

Now return to reality and consider something like the Earth-Moon system. When we integrate in the Earth-inertial reference frame and include the effect of lunar gravity, we are repeating the previous example. The Earth-inertial frame is experiencing lunar-induced gravitational acceleration similar to that of the vehicle. It would be inappropriate to include the full lunar gravity; we instead need to include only the difference between the lunar gravity experienced by the vehicle and that experienced by Earth. In other words, we need the 3rd-body perturbation to the gravitational field produced by the influence of lunar gravity (vehicle is 1st-body, Earth is 2nd-body, Moon is 3rd-body).

Consider the same simulation, but with the vehicle integrated in the Earth-Moon Barycenter inertial reference-frame. While it is certainly true that this frame is also accelerating relative to the Earth-inertial frame (the Earth-inertial frame moves around the barycenter inertial frame), we consider the Earth-Moon Barycenter frame to be "more inertial" than that of Earth-inertial. There is a hierarchy of the "inertial-ness" of the inertial frames, and it follows the same tree we have seen before:



Thus, when we integrate in the Earth-Moon Barycenter inertial frame, we are not concerned with the acceleration of this frame towards Earth, because it defines a truer inertial frame than does Earth-inertial. We consider the Earth-Moon Barycenter fixed, and the Earth-inertial and Moon-inertial frames to be accelerating relative to it. Consequently, it now IS appropriate to include the direct gravitational accelerations resulting from both Earth and Moon gravities when integrating in this frame.

Conversely, the Earth-Moon Barycenter has the same level of “inertial-ness” as the Sun-inertial frame, so the inclusion of solar gravity would have to be managed as a 3rd-body effect.

If we moved the integration frame to Solar-system Barycenter Inertial, then it would be appropriate to include the direct gravity of Sun, Earth, and Moon, since now none of these influence the “more-inertial” integration frame (Solar-system Barycenter Inertial).

To summarize, any planetary bodies that are children (descendant by one or more generations) of the integration frame will have a direct gravitational effect on the vehicle. Any planetary body whose inertial reference frame is parent, sibling, “cousin”, or any other relation that does not constitute a direct descendance from the integration frame will produce a gravitational effect that is treated as a 3rd-body perturbation.

This is all taken care of behind the scenes, but it brings up an important point – that the ephemerides tree and the reference-frame tree are expected to have the same structure. Attempts to manipulate the planetary-base of the reference-frame tree to some other organization will have dire consequences for the calculation of gravitational effects.

Chapter E Using the Mass Tree and Reference-Frame Tree

Document reference: models/dynamics/dyn_body/docs/dyn_body.pdf

models/dynamics/mass/docs/mass.pdf

models/utils/ref_frames/docs/ref_frames.pdf

E.1 Changing the Mass Tree

E.1.1 Connecting/Disconnecting Vehicles on the Mass Tree

This process is somewhat complicated, and there are *BodyAction* processes made available for performing this task. See section F.6 for details.

The behind-the-scenes variables, *composite_wrt_pbdy*, *composite_wrt_pstr*, and *core_wrt_composite* will all be updated appropriately for all mass elements when MassBody objects are attached to (or detached from) one another.

E.1.2 Adding Points of Interest to a Vehicle

It may be desirable to model some fixed point on a vehicle – such as an antenna or a reflector – so that we can compute the position or orientation of that point in some other reference frame. Perhaps the most useful points to define are those that will be used for attaching vehicles to one another. This section describes that process.

Each vehicle simulation-object should include an instance of a *DynBody*, which includes a *MassBody*, which represents the mass properties of the vehicle. We have already seen in section D.4.5 that a *MassBody* has a *MassBasicPoint* called *structure_point*, and two instances of *MassProperties* (which inherit from *MassBasicPoint*) called *core_properties* and *composite_properties*.

Inheriting from *MassBasicPoint* is the *MassPoint*, and the *MassBody* also contains an (initially empty) STL list of *MassPoint* objects – called *mass_points* – to which the user may add. This is most easily carried out by the same *MassBodyInit* body-action that is used to initialize the mass, inertia, etc. of the vehicle.

The process is quite straightforward, and is carried out in the input file.

First, state how many points are being added, and allocate that many sections of memory of type MassPointInit.

```
(based on dynamics/body_action/verif/SIM_verif_attach_mass/SET_test/RUN_110/input.py)
 45 vehicle.mass_init.num_points = 3
 46 vehicle.mass_init.points = trick.sim_services.alloc_type( 3 , "MassPointInit" )
```

Then, for each point, define its name and its position and orientation with respect to the structure-point. See section I.2 for details on options for specifying orientation.

```
 47 vehicle.mass_init.points[0].name = "front_to_back"
 48 vehicle.mass_init.points[0].position  = trick.attach_units( "m",[ -0.5, 0.0, 0.0])
 49 vehicle.mass_init.points[0].pt_orientation.data_source=trick.Orientation.InputMatrix
...
 52 vehicle.mass_init.points[0].pt_orientation.trans[0]  = [ -1.0,  0.0, 0.0]
 53 vehicle.mass_init.points[0].pt_orientation.trans[1]  = [  0.0, -1.0, 0.0]
 54 vehicle.mass_init.points[0].pt_orientation.trans[2]  = [  0.0,  0.0, 1.0]
```

By default, the orientation specified is of the mass-point with respect to the structure-point. That can be reversed with the setting of the *pt_frame_spec* value:

```
Example
vehicle.mass_init.points[0].pt_frame_spec = = trick.MassBasicPointInit.PointToStruct
```

Behind the scenes, the *MassPointInit* instances just defined are used to create new *MassPoint* instances, which are added to the *mass_points* list in the *MassBody*, and added to the Mass Tree as children of the *structure-point*.

E.1.3 Using the Mass Tree – the User Interface

E.1.3.a Extracting Relative Position and Orientation

To calculate the position and orientation of one *MassPoint* from another, the simple call *compute_relative_state* is made. It takes two reference arguments to another *MassBasicPoint*, and to a *MassPointState*.

(Note, since a *MassPoint* is a *MassBasicPoint,* this *MassBasicPoint* method is inherited for application to *MassPoint* objects as well)

```
invoking_mass_basic_point.compute_relative_state( other_point, point_state);
```

This call will compute the position and orientation of the invoking *MassBasicPoint (invoking_mass_basic_point* above) with respect to *other_point*, and populate the *point_state* value with those data.

Note that this method will ONLY work if the two points are in the same mass tree. Recall that elements in the mass tree have only information on position and orientation relative to their parent element in the tree, and not any absolute state information. Thus, it is not possible to use this method to find the relative position of two mass points on two distinct unattached vehicles. For that, we need the reference frame tree.

Exercise 18. Adding Mass Points

E.1.3.b Updating the Mass Tree

The recalculation of all of the mass properties for all bodies in all mass trees in the simulation is non-trivial, and is not preformed except as needed. When the mass properties of a *MassBody* element change behind the scenes, a flag *needs_update* gets set. When changing mass properties manually, it is essential that this flag be set by calling the *set_update_flag* method for the affected MassBody.

```
affected_mass_body.set_update_flag();
```

Note that this method will set the needs_update flag for this body and call set_update_flag on its parent body (thereby setting the flag all the way up the tree to the root of the tree).

The next step is to call the *update_mass_properties* method:

```
affected_mass_body.update_mass_properties();
```

Because the composite-properties depend on the mass properties of everything below this point in the tree, this method will descend the tree looking for the *needs_update* flag, calling this method on all of its children so flagged before trying to update itself (thereby covering the entire sub-tree before updating itself).

Notes

1. that this method **WILL NOT** update the mass properties of any *MassBody* objects higher in the tree. **It is important to call this method on the root of the mass tree.** Recall that the root body's needs update flag will be set if the *set_update_flag* has been called on any body in the tree, since that method sets the flag all the way up the tree. This interface is, perhaps, the strongest reason why it is useful to have some knowledge of the structure of the mass tree.
2. This method resets the *needs_update* flag to false to prevent repetitious calling.

E.1.3.c Extracting Tree Information

There are several tree utilities available, see the Mass Body model documentation for details on using these utilities.

Tree Output

The method *print_body* will print the tree starting with the *MassBody* of interest and descending for a specified number of levels.

The method *print_tree* calls *print_body* on the root of the tree containing the *MassBody* of interest.

Extracting Tree Relations

The method *get_root_body* returns a pointer to the *MassBody* that is at the root of the tree containing the *MassBody* of interest.

The method *get_parent_body* returns a pointer to the *MassBody* that is parent to the *MassBody* of interest.

The method *is_progeny_of* returns a boolean (true/false) indicating whether the invoking *MassBody* is in the sub-tree of the *MassBody* specified in the argument list.

Exercise 19. Modifying a Tank Mass

E.2 Changing the Reference-Frame Tree

NOTE - in the Reference-Frame model documentation, reference is frequently made to operations made by the Reference Frame Manager. Be aware that the Ephemerides Manager inherits from the Reference Frame Manager, and the Dynamics Manager inherits from the Ephemerides Manager. Therefore, when we instantiate a Dynamics Manager, we also instantiate a Reference Frame Manager. In this course, we refer to the Dynamics Manager as the entity that keeps track of reference frames; this is because it is the Dynamics Manager that is the visible front-end to the user. If you consider that when we do this, we are really referring to the Reference Frame Manager component of the Dynamics Manager, then these course materials and the model documentation are consistent.

E.2.1 Adding Vehicles to the Reference-Frame Tree

Each vehicle simulation-object should include an instance of a *DynBody*, which represents the six-degree-of-freedom dynamics of the vehicle. Each *DynBody* in the simulation automatically comes with three reference frames associated with the body. These – the *core_body*, *composite_body*, and *structure* frames – have already been discussed in section D.1.2.

By the process of registering vehicles with the Dynamics Manager, these three default reference frames associated with the *DynBody* component of a vehicle simulation-object are automatically added as children of the vehicle's integration frame. See section B.5 for details of adding a vehicle to the simulation. To specify the integration frame, use the following input-file command:

```
example
vehicle.dyn_body.integ_frame_name = "Earth.inertial"
```

Note that this frame must exist, and it must be inertial!

E.2.2 Adding Additional *BodyRefFrame* Instances to a Vehicle

It is often desirable to be able to describe the state of some point – or the relative state of some frame relative to some point – on a vehicle other than the structural origin or center of mass. For example, when modeling a lidar system, we may want the state of a reflector in the reference frame of the lidar case; or when modeling the output of an accelerometer, we may want the state of the reference frame associated with the accelerometer. Particularly when the vehicle is rotating, these state calculations can be non-trivial to perform outside the simulation.

The three default frames of a *DynBody* are all instances of the *BodyRefFrame* class. Adding an additional *BodyRefFrame* provides the ability to model additional frame states. In addition to the three default *BodyRefFrame* instances in a *DynBody*, the *DynBody* also contains an STL list of additional instances of *BodyRefFrames* instances associated with the vehicle. This list is called *vehicle_points.*

By now, you may have noticed some similarities between adding *MassPoint* elements to the *MassBody*, and adding *BodyRefFrame* elements to the *DynBody.* In fact, the processes are identical.

When adding *MassPoint* elements to a *MassBody*, we make *MassPointInit* instances, then call the *MassBody* method *add_mass_point* (through the *MassBodyInit* Body Action) which creates *MassPoint* instances and adds them to the *mass_points* list.

When *add_mass_point* is called by a *DynBody, e*verything that the *MassBody* implementation performs is included through the *DynBody MassBody* (*mass*), so the mass tree looks the same. Beyond that, we recognize that the parent body now has reference frames, so we also perform the following tasks:

- make an instance of a *BodyRefFrame* from the newly constructed *MassPoint*,
- add that frame as a child of the *DynBody* object's integration frame (if it exists)
- add the new *BodyRefFrame* to the *DynBody* object's *vehicle_points* list.
- Add a pointer to the new *BodyRefFrame* to the Dynamics Manager's list of reference frames.

Once added in such a way, the state of this reference frame will be automatically updated as long as there is some part of the simulation that subscribes to it. Note that the frame can be found with the *find_vehicle_point* method. This method is analogous to *find_mass_point*; it takes a name argument, and returns a *BodyRefFrame* (instead of a *MassPoint*). The name of the *BodyRefFrame* is copied from the name of the *MassPoint*.

E.2.3 Adding Additional *RefFrame* Instances to the Simulation

E.2.3.a Derived States

It is also frequently necessary to add non-vehicular reference frames to the environment in which the vehicle is operating. This type of reference frame are often called Derived States, and include frames such as planet-fixed, Local-Vertical-Local-Horizontal, North-East-Down, etc. We devote an entire chapter of the course to discussion of derived states, see chapter H.

E.2.3.b Stand-alone Reference Frames

In very specialized cases, additional reference frames may be added to the simulation by instantiating them, and adding them to the Dynamics Manager (aka Reference Frame Manager) one at a time. A reference frame may only be added to the manager once, and it must have a unique name when doing so. The following utilities are provided by the Dynamics Manager, more information can be found on these in the Reference-Frame Model documentation:

Add a frame

The method *add_frame* will register the frame with the Dynamics Manager, but will not add it to the reference frame tree.

Place a frame in the reference-frame tree

The method *add_frame_to_tree* adds a frame into the reference -frame tree as a child of the specified parent.

Subscribing to Reference Frames

Calling the *subscribe_to_frame* method with either the name of the frame, or a direct reference to the frame itself as an argument will add a subscription to the reference frame. As long as a reference frame is registered with the Dynamics Manager **and** placed in the tree somewhere **and** has at least one subscription, its state will be updated automatically.

Unsubscribing from a Reference Frame

Similarly, the unsubscribe_to_frame method will remove a subscription. Be very careful when using this method that only those subscriptions that have been manually added get manually removed. Otherwise, it is possible to inadvertently remove the subscription that some other model has placed on this reference frame; if the number of subscriptions incorrectly falls to zero, then a model relying on that frame being updated will have no knowledge that the frame state has stopped updating and will erroneously proceed with stale data.

E.2.4 Using the Reference-Frame Tree – the User Interface

While the state of a subscribed reference frame is updated automatically, that calculation extends only as far as generating the state of the frame relative to its parent. It is frequently necessary to find the state of one frame relative to another. Where this is a routine occurrence, the *RelativeDerivedState* (see section H.5) should be used, and the relative state will be computed automatically.

Where the relative state is required only infrequently or irregularly, the following methods can be used to force the calculation of relative states.

To extract the full state of one frame relative to another, use the *compute_relative_state* call.

```
RefFrame::compute_relative_state( frame1, frame_state)
```

This computes the state of the invoking reference frame with respect to *frame1* (in the argument). The output is written to the *frame_state* object (second argument in functional call) using the convention that:

- *state.trans.position* is expressed in *frame1*
- *state.trans.velocity* is expressed in *frame1*
- *state.rot.T_parent_this* is the transformation matrix from *frame1* to *frame2*; it is not expressed in any frame, transformation matrices are frame neutral.
- *state.rot.ang_vel_this* is expressed in *frame2.*

A similar call, to obtain position information only, is:

```
RefFrame::compute_position_from( frame1, double relative_position[3])
```

This call works the same way, the position of the invoking frame is computed relative to *frame1*, and written to *relative_position.*

Exercise 20. Simple Exercise with Mass Points

E.3 Loggable data

E.3.1.a Vehicle State

Probably the most commonly logged data of any variables in the simulation are the state of the vehicular reference frames:

```
example log-data file
for ii in range(0,3) :
  dr_group.add_variable("vehicle.dyn_body.composite_body.state.trans.position[" + str(ii) +
"]" )
  dr_group.add_variable("vehicle.dyn_body.composite_body.state.trans.velocity[" + str(ii) +
"]" )
  dr_group.add_variable("vehicle.dyn_body.composite_body.state.rot.ang_vel_this[" + str(ii)
+ "]" )
  for jj in range(0,3) :
    dr_group.add_variable("vehicle.dyn_body.composite_body.state.rot.T_parent_this[" +
str(ii) + "][" + str(jj) + "]" )
```

and similar for *core-body* and *structure* in place of *composite_body*.

E.3.1.b Vehicle Properties

The mass and inertia tensor, and the position of the center of mass relative to the structure-point (i.e. relative to, and expressed in, the structural frame). The position and orientation properties-variables also define the position and orientation of the respective body frame with respect to the structural frame.

```
example log-data file
dr_group.add_variable("vehicle.dyn_body.mass.composite_properties.mass" )
dr_group.add_variable("vehicle.dyn_body.mass.composite_properties.Q_parent_this.scalar" )

for ii in range(0,3) :
  dr_group.add_variable("vehicle.dyn_body.mass.composite_properties.position[" + str(ii) +
"]" )
  dr_group.add_variable("vehicle.dyn_body.mass.composite_properties.Q_parent_this.vector["
+ str(ii) + "]" )

  for jj in range(0,3) :
    dr_group.add_variable("vehicle.dyn_body.mass.composite_properties.T_parent_this[" +
str(ii) + "][" + str(jj) + "]" )
    dr_group.add_variable("vehicle.dyn_body.mass.composite_properties.inertia[" + str(ii) +
"][" + str(jj) + "]" )
```

and similar for the *core_properties* in place of *composite_properties.*

Chapter F Body-Actions

Document reference: models/dynamics/body_action/docs/body_action.pdf

F.1 Introduction to Body-Actions

The Body Action Model allows us to manipulate those simulation objects that represent real-world objects, such as vehicles. The model itself provides the generic framework, with many specific applications inheriting from the base-class *BodyAction.* Examples include:

- Set the mass properties of a MassBody object (*MassBodyInit* type of *BodyAction*)
- Attach and detach MassBody and DynBody objects (*BodyAttach* / *BodyDetach* types of *BodyAction*)
- Set the state of a DynBody object (*DynBodyInit* type of *BodyAction*)
- Change the frame in which a DynBody's state is propagated (*DynBodyFrameSwitch* type of *BodyAction*)

The Body Action Model directly affects data held within the MassBody and DynBody objects, and is intended to act **through the Dynamics Manager**. You have to use the middle man!

WARNING: While the individual *BodyAction::apply* methods CAN be called from without the Dynamics Manager, THIS CAN CAUSE UNEXPECTED BEHAVIOR. **DO NOT TRY IT** (unless you really know what you are doing)!!

Because the BodyAction instances are intended to handled by the Dynamics Manager, all such instances must be registered with the Dynamics Manager.

```
Example:
dynamics.dyn_manager.add_body_action( vehicle.mass_init )
```

The Dynamics Manager contains a method *perform_actions*, which cycles through all of the currently-registered *BodyAction* objects, testing for which actions are qualified for immediate application. Any that may be enacted upon are performed. Once completed, the instance is deleted from the Dynamics Manager's list.

Important consequence: *BodyAction* instances are single-use entities.

Again, it is possible to perform a *BodyAction* without the Dynamics Manager, but the complexity of trying to circumvent this single-use protection is formidable and far more elaborate than just declaring additional instances. **DO NOT TRY TO REUSE *BodyAction* INSTANCES.**

F.1.1 General Methods for Using BodyActions

F.1.1.a Instantiate Specific BodyAction sub-class Instance

We will be using the *MassBodyInit* as an illustrative example, but all *BodyAction* sub-classes follow the same process.

Making a new instance of a *BodyAction* can be performed two ways. Common to both is the inclusion of the appropriate header file in the S_define. In this example, there are three body-actions to be instantiated, so three header files are included.

lib/jeod/JEOD_S_modules/vehicle_basic.sm

```
 10 ##include "dynamics/body_action/include/dyn_body_init_trans_state.hh"
 11 ##include "dynamics/body_action/include/dyn_body_init_rot_state.hh"
 12 ##include "dynamics/body_action/include/mass_body_init.hh"
```

Then, the actual instance(s) may be created in either the S_define:

```
 24    jeod::DynBodyInitTransState   trans_init;
 25    jeod::DynBodyInitRotState     rot_init;
 26    jeod::MassBodyInit             mass_init;
```

or at the input file level:

models/dynamics/dyn_body/verif/SIM_VERIF_ATTACH_DETACH/SET_test/RUN_simple_attach_detach/input.py

```
259 attach_1_to_2 = trick.BodyAttachAligned()
```

F.1.1.b Defining the BodyAction

The appropriate data for the specific action is typically populated at the input file / Modified-data file level. For body-actions instantiated in the S_define, these data could also be defined in default data files (recall that default data is run before the input file, so body-actions instantiated in an input-file do not exist at the time default-data is processed, so default-data population of such entities is not an option).

Consider, as an example, the setting of mass. We have already covered this in section B.5.3.b.

Example

```
vehicle.mass_init.properties.mass = 1000.0
```

F.1.1.c Applying the BodyAction

If the action is to be applied immediately, no additional steps are needed. If the action is to be performed at some later time in the simulation, follow the steps outlined in section F.3.

F.2 Using Body Actions for Initialization

There are multiple aspects to initializing the bodies, and multiple options by which the initialization may be performed. Particularly for initializing the dynamic states, it is strongly recommended that body-actions be used. Behind the scenes, checks are made that the state initialization is valid before it is applied.

For example, consider a two vehicle simulation in which the dynamic state of (the frames of) vehicle A is initialized with respect to the LVLH frame associated with vehicle B. Clearly, the state of vehicle B must be initialized first, then an LVLH frame defined based on that state, and only then can the inertially-referenced state of vehicle A be generated. Without utilizing the body-actions, trying to line up the necessary function calls to accomplish this is quite the task. With the body-actions, the registered body-action will identify its dependencies and declare itself invalid until those dependencies have been met. The Dynamics Manager will keep cycling through the available body-actions, processing everything that is valid on each pass, until the support for vehicle A is available; at that point, it will be initialized and all without any additional configuration from the user.

F.2.1 Preferred Order of Initialization

While the Dynamics Manager will handle most sequencing of body-action calls, it helps to add the body-actions in vaguely the correct order. The body-actions should be processed in the following order:

1. *MassProperties* initializations
2. Set the appropriate *MassPoint* instances
3. The attachment operation *BodyAttach*
4. The dynamics state initializations (after the composite-vehicles have been joined)
5. All other actions after states of the simulation's *DynBody* objects have been set

F.3 Using Body Actions at Run-time

Run-time body-actions are typically in response to either some scheduled event (e.g. detach stage 1 from stage 2 at time *t*), or some dynamic event (e.g. switch to a different integration frame when the state exceeds some limit). The simplest way to process run-time body-actions is to register the action at initialization, but set the *activate* flag to false (the default setting is to apply immediately)

models/dynamics/dyn_body/verif/SIM_VERIF_ATTACH_DETACH/SET_test/RUN_simple_attach_detach/input.py

```
300 detach_1 = trick.BodyDetach()
...
304 detach_1.activate = False
```

The action is registered with the Dynamics Manager.

```
306 mngr.dyn_manager.add_body_action (detach_1)
```

The Dynamics Manager *perform_actions* method cycles through its list of registered and unapplied *BodyAction* instances. Any that have their *activate* flag set to true then run an additional test called *is_ready*. A body-action will only be applied when this internal *is_ready* method indicates readiness to be applied, and the method is only called when the externally-set *activate* flag is *true*.

For most body-actions, the *is_ready* method always returns *true*, but this provides one additional layer of automation allowing the body-action to verify that its pre-requisite data have been set.

For example, consider the integration-frame-switch body-action. It may have the *activate* flag set when the vehicle comes close to the optimal change-over point. When the Dynamics Manager comes across this body-action, it sees the activate flag, and knows that the body-action should run its *is_ready* method. That test takes the vehicle position-state and compares it to the optimal change-over location, and sets the returns *true* only as (and after) the vehicle passes that optimal change-over position. Now, with both indicators *true*, the body-action gets applied. If the *activate* flag never gets set, the position never gets compared and the body-action never gets applied.

Thus, a *BodyAction* can be created, initialized and registered with the Dynamics Manager all within the input file; then, when it is to be applied, flip the *activate* flag to True. If it is properly configured, it will then be available for application.

models/dynamics/dyn_body/verif/SIM_VERIF_ATTACH_DETACH/SET_test/RUN_simple_attach_detach/input.py
```
25 trick.add_read( 20    , "veh1.detach_from_2.active = True")
```

Operation of the Dynamics Manager *perform_actions* method can be run as a routinely scheduled job (strongly recommended):

models/dynamics/dyn_body/verif/SIM_VERIF_ATTACH_DETACH/S_define
```
117       (DYNAMICS, "environment") dyn_manager.perform_actions ( );
```

or as a specific call made along with the flag being set.

(Example equivalent to models/dynamics/dyn_body/verif/SIM_VERIF_ATTACH_DETACH/S_define)
```
332 detach_1.activate = True
333 dynamics.dyn_manager.perform_actions()
334 """)
```

The former method has the distinct advantage of having the method called repeatedly until the *is_ready* method indicates that the body-action is ready for application. The latter method has the advantage that the method is only run as needed and when it is needed (rather than when it is next scheduled) but should only be used when the user is confident that the body-action will process with the *is_ready* method returning true on the first time through.

F.4 Initializing Mass Properties

All vehicles should have an instance of the *MassBodyInit* body-action, instantiated either in the S_define or the input file (as already discussed). The simulation-module *vehicle_basic.sm* includes such an instance, called *mass_init*.

lib/jeod/JEOD_S_modules/Base/vehicle_basiline.sm
```
 26    jeod::MassBodyInit             mass_init;
```

We have already been using this in every simulation-building exercise up to this point. Within mass properties, we set the core-properties of the body. These are used to initialize the composite-properties, which are then updated as any other bodies get attached. Within mass properties, we define:

- the subject of the action – that is, the vehicle that is being initialized with the body-action.

  Example
  ```
  vehicle.mass_init.subject = vehicle.dyn_body.mass
  ```

- The vehicle mass (the core mass)

```
vehicle.mass_init.properties.mass = 1000.0
```

Note that the units specification is unnecessary if SI units are being used, but any other units should be converted either prior to assignment or as part of the assignment using some conversion process such as that provided by *trick.sim_services.attach_units*.

- The vehicle inertia tensor

```
vehicle.mass_init.properties.inertia[0] = [500.0,    0.0,    0.0]
vehicle.mass_init.properties.inertia[1] = [  0.0,12250.0,    0.0]
vehicle.mass_init.properties.inertia[2] = [  0.0,    0.0,12250.0]
```

By default, the inertia tensor is that of the vehicle using the body-axes with origin at the center of mass. Optionally, other axes-sets may be specified, including:

  - Use the structural axes with origin at the structure-point (*MassPropertiesInit::Struct*)
  - Use axes aligned with the structural axes with origin at the center of mass (*MassPropertiesInit::StructCG*)
  - Use a set of user-defined axes with origin at the center of mass (*MassPropertiesInit::SpecCG*). With this option, the user must also specify *inertia_orientation*, the orientation of this axes-set.
  - Use a set of user-defined axes with origin at a user-defined location (*MassPropertiesInit::Spec*). With this option, the user must also specify *inertia_orientation* (the orientation of this axes-set) and *inertia_offset* (the location of the origin of the axes-set).
  - None. Setting *MassPropertiesInit::NoSpec* means that no inertia data will be specified.

- The position of the center-of-mass mass-point with respect to the structure point

```
vehicle.mass_init.properties.position = [6.0, 0.0, 0.0]
```

Note that this position also represents the position of the core-body reference frame with respect to, and expressed in, the structural frame.

- The orientation of the center of mass mass-point with respect to the structure point

```
vehicle.mass_init.properties.pt_orientation.data_source = orient_opt.InputEigenRotation
vehicle.mass_init.properties.pt_orientation.eigen_angle = 0.0
vehicle.mass_init.properties.pt_orientation.eigen_axis  = [0.0, 1.0, 0.0]
```

Note that this designation primarily functions to represent the orientation of the core-body reference frame with respect to the structural frame. There are multiple ways to express relative orientation, see the Orientation section (I.2) for details but these include:

  - single rotation about a specified axis
  - euler-angle rotations about a set of 3-axes in some specified order
  - transformation matrix
  - quaternion.

Optionally, the orientation could specify the orientation of the structure-point relative to the center-of-mass mass-point with an additional specification of the *pt_frame_spec* variable.

Example
```
vehicle.mass_init.properties.pt_frame_spec = trick.MassBasicPointInit.BodyToStruct
```

F.5 Adding Mass Points to a Body

The details for adding *MassPoint* instances has already been covered in the discussion of the mass tree. See section E.1.2. Mass-points are added by the same *MassBodyInit* body-action that defines the other mass-properties. The same Dynamics Manager registration call that registers the mass properties initialization process serves to register the mass-points initialization process.

Exercise 21. A New Two-vehicle Simulation

Exercise 21.1 Setting up Mass Properties

F.6 Attach and Detach Vehicles

While the attach and detach processes are included in the Body Action chapter, it is not necessary to use a body-action to process an attach or detach. These methods may be called directly from another method. We will discuss the available body-actions, and point out where the body-actions make the function calls that could be made without the Body Action model. The body-action implementation of attach and detach mechanisms do provide some security that the processes are being called correctly, and do make the configuration a little easier. Where possible, the Body Action implementation should be used. The direct (non-body-action) implementation of the attach/detach mechanism is presented in section F.6.5.

Definitions

When two bodies are attached, there is a **child** body and a **parent** body. On the mass tree, the child body is made **inferior** to the **superior** parent body.

F.6.1 Effect of Attach/Detach on Properties and State (aside)

F.6.1.a Properties

When two objects attach (or detach), the core-properties of both vehicles, and the composite-properties of the child remain unchanged. The composite-properties of the parent body are changed to accommodate the additional mass and inertia, and the shift in the location of the center-of-mass resulting from the addition of mass at a location that is generally not at the old center of mass.

F.6.1.b State

The state of the composite-body, core-body and structural frames of both vehicles will, in general, be affected by the attachment (or detachment).

Conservation Laws (aside)

The processes obey the physical momentum-conservation laws (both linear and angular). During the detach process, the vehicles are simply released, and the conservation laws may be applied trivially. The attach process is more problematic. Ideally, the two vehicles would be touching before attaching, and in this case, the conservation laws would again apply perfectly. However, that is not typically possible in a simulation environment and instead, we get the vehicles "close enough", at which point the child instantaneously "snaps" into its final position, a non-physical event that complicates the conservation of angular momentum. During this snap-attach, linear momentum is conserved, and angular momentum is conserved in the frame that is centered at, and moving with, the post-attachment center of mass.

Note that if the final "snap" is a small displacement, there is no discernible difference between the ideal attaching of two adjacent vehicles and the simulated attaching over a finite displacement.

Realize that if the vehicles are erroneously snap-attached from meters (or kilometers) apart, then the departure from physical reality is extreme. While the momentum conservation laws will still be applied as detailed above, there is a very significant likelihood that the application of those physical constraints to such a non-physical scenario will result in violations elsewhere, most notably in unphysical gains in rotational kinetic energy.

Child body

The position and orientation of the frames may show a small step function during attach resulting from the snap-attach nature of the final connection. The transition during detach will be smooth.

The velocity and angular velocity will be populated from the parent body after attach.

The velocity of the child body's center-of-mass will be unchanged by the detach process. Recognize that this velocity is a function of the parent body's velocity, angular velocity, and the relative position of the child from the combined center of mass.

The child's angular velocity will be unchanged during detach.

Parent body

The position and orientation of the core-body frame should be unaffected, since, for the attach process, it is the child body that is moved during the final snap adjustment, and the detach process is a smooth transition. For the composite-body frame, the orientation is conserved, but the position will exhibit a step as the center-of-mass shifts instantaneously to accommodate the instantaneous addition to (or loss of) mass.

The velocity and angular velocity of the center of mass of the combined vehicle are both conserved (since mass is assumed constant), but since the variables represent the velocity and angular velocity of the current center of mass, and the center of mass changes instantaneously, the composite-body frame velocity and angular velocity will both exhibit step functions. The core-body velocity and angular velocity are derived from the composite-body values, since it is the composite-body values that are conserved.

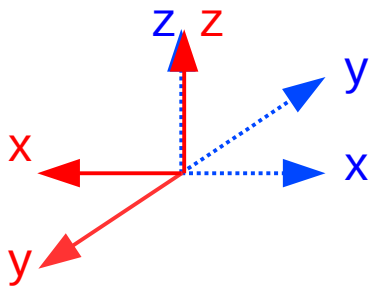
Note that prior to an attachment, angular momentum includes linear motion of bodies whose linear momentum vectors do not pass through the combined center of mass.

F.6.2 The Attachment Process

There are two body-actions for attaching vehicles together. Both derive from the general *BodyAttach* body-action, with additional information that provide the geometric details of the attachment process.

F.6.2.a BodyAttachAligned

This body-action attaches *MassBody* objects using a point-to-point attachment mechanism on each body (i.e., subject object and parent body). The two bodies must each have a *MassPoint* that will be used in the attachment. The attachment makes the two *MassPoint* objects coincident and makes the transformation between the two mass points' reference frames to be 180 degree yaw (rotation on z-axis)



Conventionally, we consider the plane-of-contact to be the y-z plane, with the outward facing normals represented by the x-axes (so the vehicle with the red/solid axes would be to the right, and the vehicle with the blue/dotted axes would be to the left in the above diagram).
To define such an attachment, the following data are specified:

- The subject *MassBody* (a.k.a. the *child* body after attaching)
- The parent *MassBody*
- The name of the *MassPoint* instance on the subject body
- The name of the *MassPoint* instance on the parent body.
- (optional) The name of the body-action.

```
dynamics/body_action/verif/SIM_verif_attach_mass/SET_test/RUN_111/input.py
 26 pt_attach1_default()
 27 components.pt_attach1.subject_point_name = "front_to_back"
 28 components.pt_attach1.parent_point_name  = "back_to_front"
```

F.6.2.b BodyAttachMatrix

This body-action attaches *MassBody* objects using a specified relative position and specified relative orientation. This method does not require the implementation of *MassPoint* instances, it positions the two vehicles based on their reference frames. The following data must be specified:

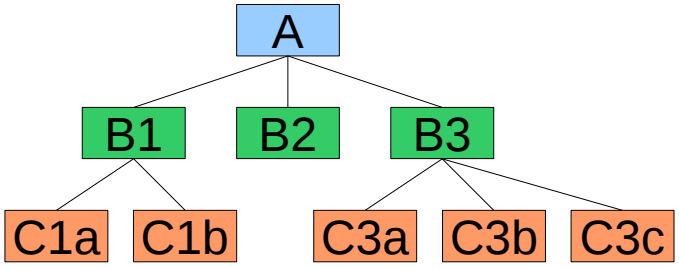
- The subject *MassBody* (a.k.a. the *child* body after attaching)
- The parent *MassBody*
- The offset position, parent-structure – to – child-structure vector, expressed in the parent-structure frame (*offset_pstr_cstr_pstr*).

- The orientation of the child structure-frame with respect to the parent structure-frame. This can be specified using any of the options available in the Orientation Model (see section I.2).
- (optional) The name of the body-action.

```
dynamics/body_action/verif/SIM_verif_attach_mass/SET_test/RUN_11/input.py
 15 attach1_default()
 16 components.attach1.offset_pstr_cstr_pstr = [ 1.0, 0.0, 0.0]
```

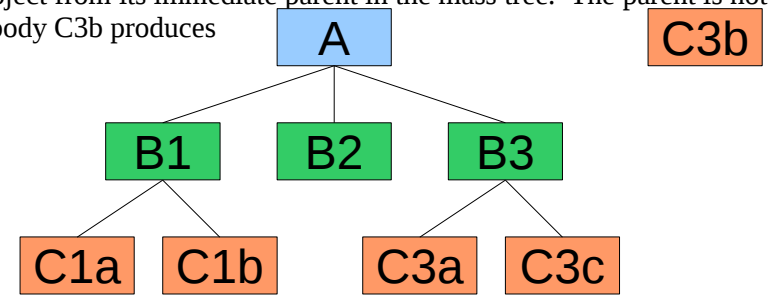
F.6.3 The Detachment Process

There are two body-actions for detaching vehicles. Both derive directly from *BodyAction*. Consider the following configuration for an illustration of the difference between the two detach processes.



F.6.3.a BodyDetach

This body-action detaches a *MassBody* object from its immediate parent in the mass tree. The parent is not specified. Applying this body-action on body C3b produces



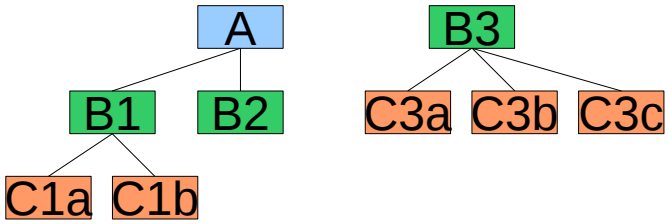
The only necessary data is the object to be detached:

```
dynamics/dyn_body/verif/SIM_VERIF_ATTACH_DETACH/SET_test/RUN_simple_attach_detach/input.py
303 detach_1.dyn_subject = veh1.dyn_body
```

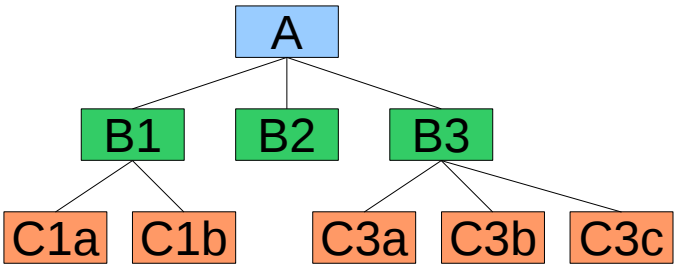
F.6.3.b BodyDetachSpecific

This body-action detaches a *MassBody* sub-tree from a specified parent by severing that part of the tree that contains the inferior *MassBody* object immediately below the superior *MassBody* object. This is useful for severing physical connections when the mass tree is not representative of those true physical connections.

Recall the discussion on the abstract nature of the mass tree in section D.4.6. In that section, we considered attaching objects C3b and A, with mass trees as illustrated:



The result was the tree:



Now, to sever that connection, we would specifically detach C3b from A, and the tree would be severed immediately below the superior object (A), returning the tree to its original form.

Simply detaching object C3b would have the effect we already studied in *BodyDetach*, leaving objects *B3*, *C3a*, and *C3c* in the mass tree headed by object *A* – even though they have no physical connection whatsoever.

The necessary data includes identification of both bodies. Note that it is not necessary to specify which is superior and which is inferior, the code will determine that for you.

dynamics/dyn_body/verif/SIM_VERIF_ATTACH_DETACH/Modified_data/attach_detach.py

```
 45  veh1.detach_from_3.subject     = veh1.dyn_body.mass
 46  veh1.detach_from_3.dyn_detach_from = veh3.dyn_body
```

F.6.4 Moving Around

The body-action *BodyReattach* takes two attached bodies, detaches them, and reattaches them in a different configuration. This only works with the position-orientation specification.

```
dynamics/body_action/verif/SIM_verif_attach_mass/SET_test/RUN_11/input.py
23 components.reattach.dyn_subject     = components.child2_body
...
25 components.reattach.offset_pstr_cstr_pstr  = trick.attach_units( "m",[ 1.5, 0.0, -2.0])
26 components.reattach.pstr_cstr.data_source =    trick.Orientation.InputEulerRotation
...
28 components.reattach.pstr_cstr.euler_sequence = trick.Orientation.Yaw_Pitch_Roll
29 components.reattach.pstr_cstr.euler_angles  = trick.attach_units( "d",[ 0.0, -90.0, 0.0])
```

Exercise 21.2 Setting the Attach and Detach Processes

Exercise 22. Review of Exercise 19

F.6.5 Attach/Detach Without the Body-Action Model

F.6.5.a Attach

The *MassBody::attach* method can be called directly with either one of two sets of arguments:

1. For the MassPoint-based attachment:
   - *const char* pointer representing the name of the child-body *MassPoint*
   - *const char* pointer representing the name of the parent-body *MassPoint*
   - *MassBody* reference representing the parent mass-body.
2. For the offset-orientation-based attachment:
   - *double[3]* (pointer) vector representing the offset position
   - *double[3][3]* (pointer) matrix representing the orientation transformation matrix
   - *MassBody* reference representing the parent mass-body.

Both methods return a *bool* to indicate whether the attach process was successful.

Notes:

1. The methods may be called from external code, or from the S_define, and the first option (but not the second) may also be called from the input file, although such practice is not recommended.

2. Python does not distinguish between arguments that represent real instances, pointers to those instances, and references to those instances. If called from the input file, it is acceptable to pass the parent-body rather than a reference to the parent-body.

3. The child body does not need to be passed in to these methods because this method is implemented by the child body.

4. Arrays, such as *double[3]* and *double[3][3]* are identified by pointers to the first element. Hence the name of the variable can be passed in (as a pointer), it is not necessary to take the address of the variable.

5. A method is identified by its class, its name, and its argument list. Although these two methods have the same name and class, they are distinct methods because they have different argument types.

F.6.5.b Detach

The *MassBody::detach* method is called for one of two options:

1. The simple case, detach this object from its parent requires no argument

2. The specific case, in which the parent is specified, takes a *MassBody* reference to the parent body.

Both methods return a *bool* to indicate whether the detach process was successful.

F.6.5.c Attach-validate and Detach-validate

The *MassBody::attach_validate* and *MassBody::detach_validate* methods are called automatically from the body-action implementation of the attach/detach processes. They verify that the proposed process is valid, looking for errors such as would be created by attaching two elements already in the same tree or detaching two elements that are not in the same tree. It should be run before attempting a manual process call. They each take a pair of arguments:

1. *const MassBody* reference to the parent body

2. *bool* indicating whether to output a message if the result is that the proposed attachment is invalid.

and return a *bool* to indicate whether the proposed process is valid.

F.7 Initializing Vehicle State

Setting the vehicle state has multiple options; as examples, the state could be expressed with respect to:

- some planetary inertial frame (we have been using this one for previous exercises),
- orbital elements referenced to some planet,

- some other vehicle body or structural frame,
- some other vehicle LVLH frame, referenced to some planet,
- some other vehicle North-East-Down frame, referenced to some planet.

Again, the body-actions can be instantiated in the S_define or in the input file. For the exercises up to this point, we have been mostly using the *vehicle_basic.sm* simulation-module, which includes instances of *DynBodyInitTransState* and *DynBodyInitRotState*. These set the translational and rotational states relative to some other frame; we have been using a planetary inertial reference frame.

F.7.1 Initializing Translational State

There are four body-actions available for initializing the translational state:

- *DynBodyInitTransState*
- *DynBodyInitLvlhTransState*
- *DynBodyInitNedTransState*
- *DynBodyInitOrbit*

For details on using these, see the Body Action Model documentation. The *DynBodyInitTransState* has already been used extensively in the exercises to this point. We will be illustrating the *DynBodyInitLvlhTransState* in section F.7.3, *DynBodyInitNedTransState* in section F.7.4 and *DynBodyInitOrbit* in section F.7.5.

F.7.2 Initializing Rotational State

There are three body-actions available for initializing the rotational state:

- *DynBodyInitRotState*
- *DynBodyInitLvlhRotState*
- *DynBodyInitNedRotState*

Again, details of using these body-actions are found in the Body Action Model documentation.

Of particular note, it is possible to initialize the rotational state of a vehicle relative to an LVLH frame associated with its own translational state, but see section F.7.3, Intializing Relative to LVLH for more details on this option.

F.7.3 Intializing Relative to LVLH

There are three body-actions associated with initializing a vehicle relative to a LVLH frame:

- *DynBodyInitLvlhTransState* initializes the translational state only (position, velocity)

- *DynBodyInitLvlhRotState* initializes the rotational state only (attitude, angular rate)
- *DynBodyInitLvlhState* initializes the translational and rotational state relative to a single LVLH frame.

Where the complete state is to be initialized off a single LVLH frame, the full-state body-action should be used. Where only the partial state is to be initialized off an LVLH, or where the translational and rotational states are to be initialized off different LVLH frames (e.g. initialize the translational state of a chaser vehicle relative to its target's LVLH, and the rotational state of the chaser vehicle relative to its own LVLH frame), then the partial (translational or rotational) options are more useful.

Determination of which states get set is configurable. It would be possible (although unlikely) to initialize the position off one LVLH frame, the velocity off another, the attiatude off a third, and the angular rate off a fourth. However, realize that any instance of these body-actions can only reference a single frame, so this task would need 4 separate instances of some sort of *DynBodyInitLvlh*State*.

An astute reader might question at this point why this particular body-action is needed at all, and why it is not possible to create a LVLH reference frame, and then initialize the state using, say, a *DynBodyInitTransState* referenced to this new LVLH frame. The answer lies in the sequencing. Because the state of the LVLH frame itself is usually (almost always) tied to the state of one of a vehicle's intrinsic frames, it is not possible to fully initialize the LVLH frame until the necessary vehicle-frame-state has been defined. But all vehicle-frame-states get set in the same call to *initialize_simulation;* that includes both the state of the frame that is initialized off the LVLH frame, and the state of the frame that defines the LVLH frame. Consequently, the LVLH frame itself must be initialized *along with* the states of those vehicle frames.

The LVLH reference frame is provided by an instance of the class *LvlhFrame*, discussed in more detail below. There are two methods by which the LVLH frame may be defined for initializing the *DynBodyInitLvlhState*, *DynBodyInitLvlhTransState*, and *DynBodyInitLvlhRotState* body-actions:

- Use an independent (user-defined) *LvlhFrame* instance in support. In this case the *DynBodyInitLvlh*State* instance populates the *LvlhFrame* data as part of the vehicle-frame-state intialization sequence; a simple *DynBodyInitTransState* would not know to perform that step.
- Use only the *DynBodyInitLvlh*State* instance in a stand-alone implementation. In this case, the initialization of the vehicle-frame-states will include the creation of a temporary *LvlhFrame* instance, its population, and subsequent destruction. Again, a simple *DynBodyInitTransState* would not know to perform that step.

Caution – it is possible – but not recommended – to mix-and-match these options. If you want an independent *LvlhFrame* instance for some purpose in the simulation, and also want to initialize from the same frame, you should use that independent frame as the reference frame for the initialization. To successfully mix-and-match requires adjustments to the sequencing because creating the independent frame and then trying to create a temporary instance of the same frame will result in a failure.

Regardless of which method is used, the process starts with the instantiation of the body-action itself:

```
models/dynamics/body_action/verif/SIM_lvlh_init/S_define
53  ##include "dynamics/body_action/include/dyn_body_init_lvlh_state.hh"
…
63   jeod::DynBodyInitLvlhState lvlh_init;
```

This body action must be populated and registered with the dynamics manager:

- Set the subject – the frame whose state is to be initialized, the reference planet name, and the name (optional) of the action

```
models/dynamics/body_action/verif/SIM_lvlh_init/Modified_data/chaser_state.py
10  chaser.lvlh_init.dyn_subject = chaser.dyn_body
11  chaser.lvlh_init.action_name = "chaser_lvlh_init"
12  chaser.lvlh_init.planet_name = "Earth"
```

- Identify the LVLH-frame; this is option-dependent and is covered specifically in those option descriptions later in this section.

- Set the state values. Note – in this case, the full state is being set off the one instance of *DynBodyInitLvlhState*.

```
 7  chaser.lvlh_init.position  = trick.attach_units( "m",[ -100.0, 25.0, 7.5])
 8  chaser.lvlh_init.velocity  = trick.attach_units( "m/s",[ 0.0, 0.0, 1.0])
 9
10  # State attitude initialization data for the chaser.
11  chaser.lvlh_init.orientation.data_source    = trick.Orientation.InputEulerRotation
12  chaser.lvlh_init.orientation.euler_sequence = trick.Orientation.Yaw_Pitch_Roll
13  chaser.lvlh_init.orientation.euler_angles   = trick.attach_units( "d",[ 0.0, 0.0, 0.0])
14  chaser.lvlh_init.ang_velocity               = trick.attach_units( "d/s",[ 0.0, 0.0, 4.5])
```

- Add the body-action to the dynamics manager

```
16  dynamics.dyn_manager.add_body_action(chaser.lvlh_init)
```

Note – The variable *reference_ref_frame_name*, used in other initialization body-actions, has no meaning when initializing the state with respect to a derived state (such as LVLH) because the name of the *reference-reference-frame* is uniquely determined by other data (such as *planet_name).*

F.7.3.a LvlhFrame

Warning – The *LvlhFrame* (as with all reference-frames) does have its own state (lvlh_frame.state, a conventional *RefFrameState*). Do not confuse the state of the LVLH reference-frame for the state of a vehicle expressed in the LVLH reference-frame.

The *LvlhFrame* model provide a **rectilinear** interpretation of LVLH, with axes defined as follows:

z: negative radial

y: negative orbital angular momentum vector

x: completes right-hand system (projection of velocity onto horizontal)

Notes:

1. The frame represents an instantaneous ‘snapshot’ of the current horizontal and vertical
2. A common misconception is to equate the x-axis with the direction of the velocity vector. In cases of near-circular orbits, the velocity vector is close to the x-axis, but that is not its definition.

The instance of *LvlhFrame* must be co-located with to another reference frame (i.e. has the same origin as the other reference frame); that other reference frame can be any other JEOD reference frame. Examples include:

- one of the intrinsic frames of a DynBody, such as the composite-body frame of a specified subject_body.
- the frame associated with a MassPoint, such as the location of a reflector.
- another planet, useful for defining the synodic rotating reference frame of something like the Earth-moon system (attaching LvlhFrame to Moon-center).

It further uses a planet, identified by name as *reference_name* to define vertical and horizontal.

F.7.3.b Initializing state relative to LVLH using LvlhFrame and Body-action

This option is typically followed when the *LvlhFrame* is needed in the simulation for other reasons, such as specifying the state of some vehicle-frame relative to some other LVLH frame (see section H.6 for details on configuring a LVLH-realtive state).

The S_define setup starts with adding an instance of *LvlhFrame* to the vehicle with which the frame will be associated. Initialization is required to build the reference-frame name; that requires knowledge of the Dynamics Manager.

```
models/dynamics/body_action/verif/SIM_lvlh_init/S_define
   54  ##include "utils/lvlh_frame/include/lvlh_frame.hh"
...
   60     jeod::LvlhFrame lvlh;
...
   70     P_ENV  ("initialization") lvlh.initialize ( dyn_manager);
```

Notice the priority on this call. *P_ENV* is very early in the initialization process; this must be complete before the frame can be used to initialize the state, which occurs with the *initialize_simulation* call with priority *P_MNGR_INIT_SIM*.

Configure this frame:

```
models/dynamics/body_action/verif/SIM_lvlh_init/SET_test/RUN_lvlh_recti_init/input.py
 47 rel_state.vehA_in_vehA_rectilvlh.subject_frame_name = "target.composite_body"
 48 rel_state.vehA_in_vehA_rectilvlh.target_frame_name  = "target.composite_body.Earth.lvlh"
```

Make this frame known to the state-initializer with the set_lvlh_frame_object method:

```
models/dynamics/body_action/verif/SIM_lvlh_init/Modified_data/chaser_state.py
 6  chaser.lvlh_init.set_lvlh_frame_object (target.lvlh)
```

Options:

1. The frame being initialized defaults to “composite_body”, but can be specified otherwise. Be careful not to build subsequent *LvlhFrame* instances for initialization purposes on uninitialized frames. For example, if structure is defined here, do not attempt to build a *LvlhFrame* on the vehicle’s composite-body state and use it for further initialization.

   ```
   Example
   vehicle.lvlh_init.body_frame_id   = “structure”
   ```

2. The LVLH initialization defaults to rectilinear LVLH. For initialization in circular-curvilinear coordinates, set the *lvlh_type* variable

```
Example
chaser.lvlh_init.lvlh_type = trick.LvlhType.CircularCurvilinear
```

Valid options for initialization are:

1. Rectilinear (default)
2. CircularCurvilinear

F.7.3.c Initializing state relative to LVLH using Body-action only

This option is typically followed when there is no other use for that particular *LvlhFrame* that is to be used for intialization. In this case, the LVLH frame can be constructed temporarily as part of the initialization.

For this option, the body-frame whose state is being initialized, and the vehicle whose state defines the LVLH frame must be specified:

```
vehicle.lvlh_init.body_frame_id   = “composite_body”
vehicle.lvlh_init.ref_body_name = “other_vehicle”
```

Note – in the case that the vehicle is providing its own LVLH frame for rotational state-intialization (*DynBodyInitLvlhRotState*), the *ref_body_name* does not need to be set; it defaults to “self”.

F.7.3.d Initializing state relative to LVLH using LvlhFrame and old syntax

It is possible --- though not recommended --- to implement this body-action using the old syntax when an equivalent instance of *LvlhFrame* is also desired. In this case, it is imperative that the permanent *LvlhFrame* instance *not* exist when the temporary one is created and destroyed. To accomplish this, change the priority on the initialziation of *LvlhFrame* from P_ENV to P_DYN. This example is included for information only; if you have an instance of *LvlhFrame*, you should use it as described above.

```
models/dynamics/body_action/verif/SIM_lvlh_init/S_define
   54  ##include "utils/lvlh_frame/include/lvlh_frame.hh"
...
   60     jeod::LvlhFrame lvlh;
...
   70     P_ENV  ("initialization") lvlh.initialize (
   71                                               dyn_manager);
   70     P_DYN  ("initialization") lvlh.initialize (
   71                                               dyn_manager);
```

F.7.4 Initializing Relative to NED

Initializing relative to North-East-Down follows a very similalr syntax to that described in section F.7.3.c, *Initializing state relative to LVLH using Body-action only*.

F.7.5 Initializing with Orbital Elements

The *DynBodyInitOrbit* body-action is used to initialize a vehicle state from orbital elements.. There are several variables which can be used to specify the orbital shape, and position of a vehicle on that orbit. These are presented in the table below, with descriptor and corresponding variable to be assigned as part of the initialization data input.

1. semi-major axis *semi_major_axis*
2. semi-parameter or semi-latus rectum *semi_latus_rectum*
3. eccentricity *eccentricity*
4. inclination *inclination*
5. longitude of ascending node *ascending_node*
6. perifocal altitude *alt_periapsis*
7. apofocal altitude *alt_apoapsis*
8. argument of periapsis *arg_periapsis*
9. time since periapsis *time_periapsis*
10. orbital radius *orb_radius*
11. radial component of velocity *radial_vel*
12. true anomaly *true_anomaly*
13. mean anomaly *mean_anomaly*
14. argument of latitude *arg_latitude*

The following can also, in theory, be used, but they are not a part of the *DynBodyInitOrbit* class

15. (momentary) velocity magnitude *vel_mag*
16. eccentric / hyperbolic / parabolic anomaly *orbital_anom*
17. mean motion *mean_motion*
18. orbital energy *orb_energy*
19. orbital angular momentum *orb_ang_momentum*

Not all of these are independent, a subset will completely define the orbital trajectory.

There are eight sets that JEOD has identified that are typically the most commonly used sets for uniquely defining an orbit. These are (numbers in parentheses refer to the entries in the list above):

- *SmaEccIncAscnodeArgperTimeperi* (1,3,4,5,8,9)
- *SmaEccIncAscnodeArgperManom* (1,3,4,5,8,13)
- *SlrEccIncAscnodeArgperTanom* (2,3,4,5,8,12)
- *IncAscnodeAltperAltapoArgperTanom* (4,5,6,7,8,12)
- *IncAscnodeAltperAltapoArgperTimeperi* (4,5,6,7,8,9)
- *SmaIncAscnodeArglatRadRadvel* (1,4,5,14,10,11)
- *SmaEccIncAscnodeArgperTanom* (1,3,4,5,8,12)
- *CaseEleven* (4,5,6,7,8,12)

Others can be added when it is possible to implement the algorithm for extracting the inertial state from the collection of elements. See the Body Action Model documentation for details.

To initialize with orbital elements, define an instance of the *DynBodyInitOrbit* body-action, decide which set of orbital elements are to be specified, and define the appropriate values:

```
Example
vehicle.orb_init.set = trick.DynBodyInitOrbit.SmaEccIncAscnodeArgperTimeperi
vehicle.orb_init.semi_major_axis  =  trick.sim_services.attach_units("km", 6800)
...
```

Additionally, two more values must be specified – the *planet_name* and *orbit_frame_name.* The *planet_name* must be that of a registered planet; the *orbit_frame_name* may be entered as either:

- That part of the frame name that follows on from *planet_name.* For example, *orbit_frame_name = "inertial"*, coupled with *planet_name = "earth"* will identify the *orbit_frame* as that with name "*Earth.inertial*"
- The full name of the frame. For example, *orbit_frame_name = "Earth.inertial"*. Note that this will only work if the prefix on the *orbit_frame_name* (i.e. *Earth*) matches the *planet_name.*

```
Example
vehicle.orb_init.orbit_frame_name = "inertial"
vehicle.orb_init.planet_name = "Earth"
```

Note – the *orbit_frame_name* identifies the frame in which the orbital elements are measured. **It must be co-located with the inertial frame (of the same planet),** but may have a relative orientation. As an illustrative example, the longitude of ascending node is measured from the x-axis (towards the y-axis) to the point at which the orbit breaks the x-y-plane in a positive z-direction. With differently oriented axes, the x-axis, indeed the x-y-plane can be very different, resulting in frame-specific values. Some of the other values, such as the semi-major-axis, are frame independent.

Finally, an inconvenient issue that was discovered post-release is that the value *reference_ref_frame_name* must be set to a valid reference-frame name. The *reference_ref_frame* that is typically obtained from the *reference_ref_frame_name* value will be set to the planet-centered-inertial frame associated with *planet_name*, and *reference_ref_frame_name* will be completely ignored. Nevertheless, it must be set to something valid.

```
Example
vehicle.orb_init.reference_ref_frame_name = "Earth.inertial"
```

F.7.6 Initializing Combined States and Partial States

Where the same method is to be used for translational and rotational state, the combined body-actions may be used,e.g. *DynBodyInitState.*

The non-redundant data needs to be set for both translational and rotational (so, for example, planet-name does not need to be set for both, but position and orientation do need to be set).

It is also possible to initialize, say, position using one body-action, and velocity using another. The variable *set_items* defines what the body-actions initialize, and takes a default value that is body-action specific. For example, the *DynBodyInitLvlhState* defaults to all values (*RefFrameItems::Pos_Vel_Att_Rate*), while the *DynBodyInitLvlhTransState* defaults to translational-only values (*RefFrameItems::Pos_Vel*)

```
Example
vehicle.lvlh_init.set_items = RefFrameItems::Pos
vehicle.trans_init.set_items = RefFrameItems::Vel
```

Alternative specifications are found in the API document in the *Items* enumeration found within the *RefFrameItems* class.

Exercise 22.1 Setting the Vehicle States and Testing

Exercise 23. Initializing with Orbital Elements

Exercise 24. Initializing from LVLH

F.8 Integration Frame Switch

There are two body-actions provided for switching integration frames:

- SwitchOnApproach
- SwitchOnDeparture

These body-actions will identify when the vehicle's position state is optimal for the switch to the other reference frame, with different tests being applied for whether the vehicle should switch *to* a new frame as it approaches it, or switch from an old frame as it departs from its vicinity. When using this body-action it is very, very strongly recommended to schedule the Dynamics Manager *perform_actions* routine, rather than relying on a single call when the body-action is activated.

F.9 User-defined Actions

As with most JEOD models, the Body Action Model was written with the intention that it could be extended to provide additional functionality as needed. The details of how to do this are provided in the Body Action Model documentation, and a quick overview of what is involved is provided here.

All classes that derive from BodyAction must:

1. Forward the *initialize* call to the immediate parent object if overwritten. Typically you should:
   (a) Check for errors
   (b) Forward the *initialize* call to the parent
   (c) Perform class-specific initializations
2. Query the readiness of the parent class if overriding the *is_ready* function

3. Forward the *apply* call to the immediate parent object

Chapter G Interactions with the Environment

With the exception of gravity, the vehicular interactions provided in JEOD are interactions with the surface of the vehicle. These include:

- Aerodynamic Drag
- Radiation Pressure
- Surface-surface contact with another vehicle.

This chapter covers those interactions, and the details of the surface model that is heavily utilized in generating those interactions. Since the atmosphere also feeds into the calculation of the aerodynamic drag, the process for adding a planetary atmosphere is also included in this chapter.

G.1 Vehicle Response to Gravity

For information on configuring a vehicle to respond to gravitational fields, see section C.2.3.

G.2 Adding an Atmosphere

Document reference: *models/environment/atmosphere/docs/atmosphere.pdf*

G.2.1 Introduction

JEOD provides a general framework for representing a planetary atmosphere, and a specific implementation of the MET (Marshall Engineering Thermosphere) Earth-atmosphere model, with wind model.

The structure of the model has similarities with the gravity manager. Both provide a global model that is applicable to all vehicles in the vicinity of the planet, and evaluated only in the vicinity of a vehicle.

In the case of the Atmosphere Model, there are two primary components – the *Atmosphere*, and the *AtmosphereState*. The *Atmosphere* provides the global model of the atmosphere, and the *AtmosphereState* provides the specific details of the atmosphere at a particular location. Typically, the *Atmosphere* object resides within the planet simulation-object, while the vehicle simulation-object has the *AtmosphereState* object.

With the implementation of the MET atmosphere, JEOD extends the base-classes *Atmosphere* and *AtmosphereState* to add the derived classes *METAtmosphere* and *METAtmosphereState*.

Note that *Atmosphere* is an abstract class; only the *METAtmosphere* (or other derived class not furnished by JEOD) can be instantiated. Conversely, the *AtmosphereState* is instantiable and is used in most applications.

AtmosphereState provides the following atmospheric parameters:

- temperature
- mass-density

- total pressure
- wind velocity (3-vector)

METAtmosphereState adds the following

- Exospheric temperature
- Mean molecular weight
- number density of elemental molecules such as Nitrogen-2, Oxygen-2, Hydrogen, etc.

Where these additional variables are not necessary (the majority of dynamic simulations), the base-class *AtmosphereState* can be used with the *METAtmosphere* atmosphere. This configuration will be illustrated in the course.

G.2.2 S_define Setup for Atmosphere (MET-atmosphere)

G.2.2.a Instantiation

The instance of *METAtmosphere*, and the optional addition of the wind-velocity model are typically included in the planet simulation-object.

```
lib/jeod/JEOD_S_modules/Base/earth_GGM02C_MET.sm
 10 ##include "environment/atmosphere/base_atmos/include/wind_velocity.hh"
 11 ##include "environment/atmosphere/MET/include/MET_atmosphere.hh"
...
 25    jeod::WindVelocity         wind_velocity;
 26    jeod::METAtmosphere        atmos;
```

G.2.2.b Default Data

Both the Wind Velocity model and the MET-atmosphere model come with default data files. For Wind-velocity, there is only one file, but there are atmospheric configuration files that correspond to different intensities of solar activity, representing minimum, mean, and maximum intensity.

Instances of these default data must be declared, and the default data jobs run.

```
lib/jeod/JEOD_S_modules/Base/earth_GGM02C_MET.sm
 29    jeod::WindVelocity_wind_velocity_default_data            wind_velocity_default_data;
 30    jeod::METAtmosphere_solar_mean_default_data              atmos_default_data;
...
 49       ("default_data") wind_velocity_default_data.initialize ( &wind_velocity );
 50       ("default_data") atmos_default_data.initialize ( &atmos );
```

G.2.2.c Dependencies

The MET atmosphere model has a time-dependence, relying on the UTC clock being available. The simulation-object must, therefore, receive the UTC clock from the time simulation-object. The process for accessing data in one simulation-object from another is identical to that seen before.

The private reference is created:

lib/jeod/JEOD_S_modules/Base/earth_GGM02C_MET.sm

```
 20   protected:
...
 22    jeod::TimeUTC      & utc;
```

The class constructor receives an argument of the appropriate type:

```
 32    Earth_GGM02C_MET_SimObject(
...
 34         jeod::TimeUTC      & utc_in,
...
 37         jeod::TimeGMST     & gmst_in)
```

The constructor initialization list populates the local value with that passed in:

```
 44         utc(utc_in),
```

The call to instantiate this simulation-object receives the UTC clock:

lib/jeod/JEOD_S_modules/earth_GGM02C_MET_RNP.sm

```
  2 Earth_GGM02C_MET_SimObject earth (env.gravity_manager,
...
  4                        jeod_time.time_utc,
...
  7                        jeod_time.time_gmst);
```

The wind-velocity model has no dependencies.

G.2.2.d Initialization Calls

The method *update_time* should be run at initialization to be sure that the atmosphere is properly configured before any forces are generated. This method takes a reference to the UTC clock.

lib/jeod/JEOD_S_modules/Base/earth_GGM02C_MET.sm

```
 55       P_ENV ("initialization") atmos.update_time( utc );
```

The wind-velocity model does not require any initialization.

G.2.2.e Run-time Calls

The same *update_time* call should be made as an environment-class job at some reasonable update rate.

```
lib/jeod/JEOD_S_modules/Base/earth_GGM02C_MET.sm
 60       (DYNAMICS, "environment") atmos.update_time( utc );
```

The wind-velocity model does not require any run-time updates.

G.2.3 S_define Setup for Atmosphere State

The Atmosphere State provides a position-specific state for the general Atmosphere and, if the wind-velocity model is included in the simulation, the wind-velocity.

G.2.3.a Instantiation

The instance of *AtmosphereState* (or *METAtmosphereState*) is typically included in the vehicle simulation-object. Where the aerodynamics interaction is provided its own simulation-object, this may be also be found in that object. Remember, use *METAtmosphereState* only when needing the additional data it provides, otherwise use the *AtmosphereState*.

```
lib/jeod/JEOD_S_modules/Base/vehicle_atmosphere.sm
  8 ##include "environment/atmosphere/include/atmosphere.hh"
...
 21    jeod::AtmosphereState       atmos_state;
```

G.2.3.b Dependencies

The atmosphere-state depends on the atmosphere configuration, on the position of the vehicle in the planet-fixed reference frame (expressed in elliptical coordinates), and optionally on the configuration of the wind-velocity.

Atmosphere Model

The atmosphere model can be passed in from the planet simulation-object using the standard process for accessing data in one simulation-object from another.

Wind Velocity Model (optional)

Similarly, the wind-velocity reference can be passed in, but only if the wind-velocity model is instantiated.

Planet-Fixed Position

The atmosphere-state depends on knowing the position of the vehicle in the planet-fixed reference frame, expressed in elliptical coordinates. We will see more on the planet-fixed reference frame in section H.4. This is typically found in either:

1. the vehicle simulation-object or,

2. in a derived-state or relative-kinematics (a.k.a. Relkin) simulation-object; this is more common when there are multiple derived-state instances in a simulation

The *vehicle_atmosphere* simulation module includes its own instance of the plant-fixed position.

```
lib/jeod/JEOD_S_modules/Base/vehicle_atmosphere.sm
 10 ##include "dynamics/derived_state/include/planetary_derived_state.hh"
...
 22    jeod::PlanetaryDerivedState pfix;
```

If the value comes from another simulation-object, it must be passed in using the same method already described for accessing data in one simulation-object from another. Note that only the output of the *PlanetaryDerivedState* is required. This is a *PlanetFixedPosition* class, found at, for example, *pfix.state*.

```
Example
private:
jeod::PlanetFixedPosition  & pfix_state;
...
VehicleSimObject vehicle( ..., derived_states.earth_pfix.state, ... )
```

Note that the planet-fixed frame has a meaningful orientation only when the appropriate orientation model is running (for Earth, that would be RNP). Without the RNP implementation, Earth's planet-fixed frame defaults to being oriented with the inertial frame, and the implementation of the MET atmosphere will be incorrect.

G.2.3.c Initialization Calls

If the planet-fixed state is instantiated in this simulation object, it must be updated. If it is instantiated elsewhere, it may be updated here, but is generally not necessary unless it is not known whether it is being updated in its own simulation-object. The state must be current before the atmosphere-state is updated. Because this is a derived-state, the vehicle must be fully initialized beforehand, so is given priority *P_BODY.*

The initialization of the atmosphere-state and wind-velocity are state-dependent, so are given an even lower priority (*P_DYN*) than the planet-fixed state update.

```
lib/jeod/JEOD_S_modules/Base/vehicle_atmosphere.sm
 37       P_BODY ("initialization") pfix.update();
 38
 39       P_DYN  ("initialization") atmos_state.update_state( &atmos, &pfix.state);
 40       P_DYN  ("initialization") atmos_state.update_wind( &wind,
 41                                       dyn_body.composite_body.state.trans.position,
 42                                       pfix.state.ellip_coords.altitude);
```

G.2.3.d Run-time Calls

The same calls are made as environment-class calls, with the important caveat: **while the priority can be set on the initialization to ensure that the *pfix* state is correct and up-to-date, the same cannot**

be done for run-time calls. It is important to ensure that the *pfix* state is calculated earlier in the S_define than the *update_state* and *update_wind* calls.
Note that this is really only an issue; if the *pfix* state is in another simulation-object, that object must exist earlier than the vehicle simulation-object in order to be passed into it in the first place. If the *pfix* object exists in this simulation-object, it is not terribly difficult to make sure that the *pfix.update* call comes first.

lib/jeod/JEOD_S_modules/Base/vehicle_atmosphere.sm

```
 47       (DYNAMICS,"environment") pfix.update();
 48
 49       (DYNAMICS, "environment") atmos_state.update_state( &atmos, &pfix.state);
 50       (DYNAMICS, "environment") atmos_state.update_wind( &wind,
 51                                       dyn_body.composite_body.state.trans.position,
 52                                       pfix.state.ellip_coords.altitude);
```

G.2.4 Input File Setup

The default data files populate the Atmosphere and WindVelocity models with basic data that is only applicable to Earth. The models are expecting position data relative to the Earth-fixed reference-frame, but all of the data is passed at the command line. There is nothing to connect behind-the-scenes.

The default-data settings can be overwritten in the input file. For the Atmosphere model, the geomagnetic index, the formulation (Ap or Kp), and solar radio nose parameters can be set. The wind-velocity model has more flexibility, with the ability to divide the atmosphere into any number of layers, and set the rotation rate of each layer. *See models/environment/atmosphere/data/src* for the default settings, and the corresponding variable names.

If the necessary planet-fixed state is being defined as part of the atmosphere implementation, that does need to be defined in the input file:

Example

```
vehicle.pfix.reference_name = "Earth"
```

There is an *active* flag in the *AtmosphereState* model that can be set to false to stop the atmosphere state from updating.

Example

```
vehicle.atmos_state.active = False
```

G.2.5 Logging Data

The atmospheric parameters are found in the atmosphere-state:

Example

```
dr_group.add_variable("vehicle.atmos_state.density")
dr_group.add_variable("vehicle.atmos_state.pressure")
dr_group.add_variable("vehicle.atmos_state.temperature")
```

Exercise 25. Add an Atmosphere

G.3 The Surface Model

Document reference: *models/utils/surface_model/docs/surface_model.pdf*

G.3.1 Introduction and definitions

The Surface Model is used to facilitate modeling of surface-dependent interactions, such as aerodynamic drag, radiation pressure, and contact forces.

G.3.1.a Surfaces and Facets

A **surface** comprises multiple **facets**, which together represent the exterior physical characteristics of a vehicle. A surface (of facets) is used to create one or more **interaction-surfaces** (of **interaction-facets**); each interaction-surface is used to model the specific interaction between the exterior surface of a vehicle and some environmental state.

For example, a radiation-surface is an interaction-surface used to model the interaction between a vehicle surface and the radiation field. The radiation-surface is created behind the scenes from the user-defined surface.

The two concepts – surface and interaction-surface – are not related by inheritance. A surface is not an interaction surface, and an interaction-surface is not a surface. The surface parameters are used in the creation of the interaction-surface, and their relationship ends there. The same can be said of the relation between facets and interaction-facets.

Facets are surface-sections with some known topology (e.g. flat). **Interaction-facets** are created from facets, and together form the interaction-surface. In the current release of JEOD:

- The radiation-surface (Radiation Pressure Model) can use flat plates only. The Radiation Pressure Model can also handle spherical surface topology, but through an implementation that is independent of the surface model.
- The aerodynamic-surface (Aerodynamics Model) can use flat plates only. The Aerodynamics Model can also handle unspecified topology with the use of a coefficient of drag or ballistic coefficient, but again this is independent of the surface model.
- The contact-surface (Contact Model) can use circular-flat-plates and cylinders.
- All interaction-surfaces have the ability to use any facet topology, if the interaction with that topology can be defined. So, for example, by defining the aerodynamic response of a truncated-cone, we could make a truncated-cone-shaped facet, and model the aerodynamic response of certain vehicles with such a single facet.

G.3.1.b Parameters and Factories

Each facet is defined by a collection of parameters. There are two types of parameters – those which are of general applicability (typically associated with geometry and state), and those which are interaction-specific (typically associated with the surface material).

A facet-factory will take a facet, and use its general parameters, and the interaction-specific parameters to make an interaction-facet. A different facet-factory will make a different interaction-facet from the same facet, the same general parameters, and a set of interaction-specific parameters that are relevant to a different interaction.

A surface factory will utilize the facet-factories to create an interaction-surface from the surface.

G.3.2 Creating a Surface

Each surface model must first be instantiated. Typically, this is done in the vehicle simulation-object, with one surface per vehicle. For example:

```
##include "utils/surface_model/include/surface_model.hh"
...
    jeod::SurfaceModel          surface;
```

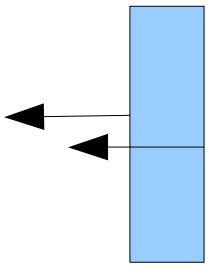
A surface is defined by a collection of facets. Facets are typically defined in Modified_data files. Once defined, they may be used in the creation of multiple interaction-facets (typically one interaction-facet for each desired interaction).

G.3.2.a Mathematics of Facets

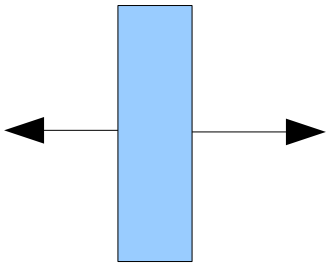
1. Facets are one-sided (so a flat, thin plate has two facets).
2. Facets have a normal vector that identifies the planar face.
3. Surfaces are generally assumed closed, universally convex, with facet normals pointing away from the surface. The surface model provides no capability for self-shadowing.

Example – a thin, flat plate comprising two facets

Incorrect, both sides of the vehicle point the same way. If this vehicle were moving to the right through a static atmosphere, it would experience no drag force.



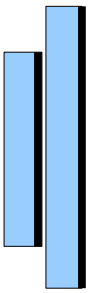
Correct, these two sides have normals that define the exterior of the vehicle. If this vehicle were moving to the right through a static atmosphere, the right-hand side would experience a drag force.



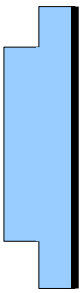
4. As a direct consequence of surfaces being closed and universally convex, it is recommended that when two vehicles attach, a new surface is provided for the composite vehicle. Maintaining two surfaces will generally violate the assumed geometry, and generate forces in self-shadowed regions.

Example – a compound vehicle moving to the right through a static atmosphere.

Incorrect – two surfaces are maintained. The forward-facing facet (heavy line in diagram) of the smaller vehicle will still generate a drag force.



Correct – new surface defined. The only forward-facing facet is now that of the larger vehicle.



5. Generic facets are spatially undefined. While the collection of facets conceptually represents a solid body, only the area, position, and orientation are defined. The dimensions, shape, and edges are not.

6. Specific geometric facets (found in the contact model) do have dimensions and shape. These can be used as a prototype for creating geometric facets for other interactions.

G.3.2.b The General Parameters

The type of facet(s) being used must be included in the S_define, this allows the instantiation in the input / Modified-data files.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/S_define
 28 ##include "utils/surface_model/include/flat_plate.hh"
```

For each facet, we create an instance of the facet type, and assign its facet-specific parameters:

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/shuttle_plate_orbiter.py
 24 flat_plate = trick.FlatPlate()
...
 26 flat_plate.position = trick.attach_units("in", [1255.0 , 0.0, 383.4 ])
 27 flat_plate.area  = trick.attach_units("m2", 119.4454385 )
 28 flat_plate.normal = [1.0 , 0.0 , 0.0  ]
...
 30 flat_plate.temperature = 70
```

Here, the position is specified in the vehicle structural reference frame, and the normal is typically the outward normal from the vehicle.

Additionally, all facets have a *name*, and a *mass_body_name* (the name of the *MassBody* object of whose surface the facet forms a part). The *name* variable is often only specified when the surface is to be used for generating a Contact Surface. The *mass_body_name* is often only used with either the Contact Model again, or when the surface is articulating. Both of these models are covered later in this chapter. The *name* is unconstrained, the *mass_body_name* must match the name of a *MassBody* already registered with the Dynamics Manager.

```
Example
flat_plate.name = “Front Lead face, upper right corner”
flat_plate.mass_body_name = “vehicle”
```

G.3.2.c The Interaction-specific Parameters

Each facet is also given a set of material characteristics that will define its interaction behavior. In general, the same material is used for more than one facet on a vehicle, so we associate a material with each facet, and the parameters with the material, rather than repetitively populating multiple parameters on every facet.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/shuttle_plate_orbiter.py
 29 flat_plate.param_name = "flat_plate_material"
```

Each interaction has its own set of parameters, and each material can be assigned to a set for each interaction. Any parameter sets that are being used must be included in the S_define:

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/S_define
 24 ##include "interactions/aerodynamics/include/flat_plate_aero_params.hh"
```

Then the parameters can be defined. Remember to name the parameter set; it is this name that will be used to match this parameter set to the facet material.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/shuttle_plate_orbiter.py
 84 params = trick.FlatPlateAeroParams()
...
 86 params.drag_coef_norm = 2
 87 params.drag_coef_tang = 2
 88 params.drag_coef_spec = 2
 89 params.drag_coef_diff = 2
 90 params.epsilon = 0.0
 91 params.coef_method = calc_coeff
...
 93 params.name = "flat_plate_material"
```

The simulation we are using as an example could have also added radiation-pressure parameters, then the same facet, made out of “flat_plate_material” could be used to create both an aerodynamic-surface and a radiation-surface:

```
Example
##include "interactions/radiation_pressure/include/radiation_params.hh"
```

```
Example
params = trick.RadiationParams()
...
params.albedo = 0.5
...
params.name = "flat_plate_material"
```

G.3.2.d Loading up the Surface and the Factories

All facets must be added to a surface. All parameter sets must be loaded into the appropriate factory.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/shuttle_plate_orbiter.py
 32 aero_test.surface.add_facet(flat_plate)
...
 42 aero_test.surface.add_facet(flat_plate)
...
 52 aero_test.surface.add_facet(flat_plate)
...
 95 aero_test.surface_factory.add_facet_params(params)
```

Note – the *aero_test* simulation object is defined in the S_define:

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/S_define
 44    jeod::SurfaceModel surface;
...
 46    jeod::AeroSurfaceFactory surface_factory;
```

G.3.3 Creating an Interaction Surface

The following are necessary to make an interaction-surface:

- A Surface Model surface
- An interaction-surface model
- An interaction-specific surface factory
- The necessary headers. Remember to include the facet-parameters header; these instances are created at the input-file level.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/S_define
 20 ##include "utils/surface_model/include/surface_model.hh"
...
 22 ##include "interactions/aerodynamics/include/aero_surface.hh"
 23 ##include "interactions/aerodynamics/include/aero_surface_factory.hh"
 24 ##include "interactions/aerodynamics/include/flat_plate_aero_params.hh"
...
 44    jeod::SurfaceModel surface;
 45    jeod::AeroSurface aero_surface;
 46    jeod::AeroSurfaceFactory surface_factory;
```

The surface should be populated with facets, and the surface-factory should be populated with material-types (or parameter-sets, they are the same thing). With those populated, the factory can make the interaction surface, in this case making *aero_surface* from *surface*. The surface and interaction-surface are passed to the factory as pointers. This process is largely independent of the rest of the simulation initialization, but most closely fits with the *P_BODY* priority.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/S_define
 52       ("initialization") surface_factory.create_surface(&surface, &aero_surface);
```

G.3.4 Using the Interaction Surface

The application of the interaction surface to particular interactions is explored in the respective sections on Aerodynamic drag, Radiation Pressure, and Contact Model. These follow an aside on making facets thermally active, and on collecting forces and torques.

Exercise 26. Setting up a Surface Model

G.4 The Thermal Rider Model

Document reference: *models/interactions/thermal_rider/docs/thermal_rider.pdf*

The Thermal-Rider Model adds capabilities to an interaction model in the case that the interaction either depends on, or affects, the temperature of the surface. The model provides the thermal modeling capabilities that – for the sake of efficiency – the surface model does not routinely provide.

If temperature modeling is not relevant, this model should not be added. It is useful, for example, for modeling the temperature differential between the surface and free-stream in the free-molecular flow of aerodynamics, or for modeling the radiation-pressure force from thermal emissions.

To simply model temperature without any forces, the simplest solution is to add the Radiation Pressure Model, and use it with forces and torques turned off. The Radiation Pressure Model already includes a full implementation of the Thermal-Rider Model.

Adding the Thermal-Rider model is a two-stage process, the facets must be be provided with the *ThermalFacetRider*, and the thermal integration called with the *ThermalModelRider::update* method.

G.4.1 Thermal Model Rider

The *ThermalModelRider* could be added at the S_define level, or in the code itself. The Radiation Pressure Model requires the Thermal Rider Model to function, so an instance of the *ThermalModelRider* is included as a component in the *RadiationPressure* class:

```
models/interactions/radiation_pressure/include/radiation_pressure.hh
 58 class RadiationPressure          /*------DYNAMIC RADIATION MODEL ------------- */ {
 93   ThermalModelRider thermal; /* --
```

and the *update* method called from the *RadiationPressure* update method:

```
models/interactions/radiation_pressure/src/radiation_pressure__surface_model.cc
 115 RadiationPressure::update_facet_surface (
...
140    thermal.update (surface_ptr);
```

Notice that the *ThermalModelRider::update* method takes a pointer argument to the vehicle surface.

The Thermal Model Rider contains two flags:

- *active* indicates whether the model is currently active
- *include_internal_thermal_effects* indicates whether to include thermal sources from the vehicle itself.

If *include_internal_thermal_effects* is set to true, the surface passed to the *thermal.update* function needs a definition of the method *accumulate_thermal_sources.* Note that this method defaults to a null implementation if not provided. The bulk of the functionality is provided by the *ThermalFacetRider::accumulate_thermal_sources* method, see the implementation in *RadiationSurface* for more clarity.

G.4.2 Thermal Facet Rider

To add thermal capability to the facets, a new plate-type is defined. For example, a *FlatPlate* type of facet with the Thermal Rider added is called a *FlatPlateThermal* type of facet, the difference is the addition of a *FacetThermalRider* instance to the header file:

```
models/utils/surface_model/include/flat_plate_thermal.hh
 49 class FlatPlateThermal : public FlatPlate {
...
 64    ThermalFacetRider thermal; /* --
```

When defining thermal facets, the following values must be additionally defined (these will vary from facet-to-facet, even when those facets are made of the same material):

- *thermal.active* makes the temperature of the facet variable
- *thermal.thermal_power_dump* provides the rate at which thermal energy is transferred to the facet from the body (e.g. for radiators). If not specified, this is assumed to be zero. The value of this parameter is assumed to be a number of Watts.

Note that this in this case, *thermal* is the instance of the *ThermalFacetRider* that is contained within the definition of the *FlatPlateThermal* facet.

```
  8 fpt = trick.FlatPlateThermal()
  9 fpt.thisown = 0
...
 15 fpt.thermal.active = True
 16 fpt.thermal.thermal_power_dump = 0.0
 17
 18 radiation.surface.add_facet(fpt)
```

When defining the parameters for thermal facets, an instance of *ThermalParams* must be added to the interaction-specific parameter class:

```
models/interactions/radiation_pressure/include/radiation_params.hh
 48 class RadiationParams : public FacetParams {
...
 69   ThermalParams thermal; /* --
```

The following values must then be additionally defined (these are material properties, so are defined with the material parameters):

- *thermal.emissivity* is the ability to radiate energy away ($P = \epsilon \sigma A T^4$, where ϵ is the emissivity)
- *thermal.heat_capacity_per_area* is the heat capacity (note – not specific heat capacity) of the material per unit area. It is assumed expressed in units of $J\ K^{-1}\ m^{-2}$. This is unique among material properties because it is affected by the material thickness (e.g. a thick ceramic will have a higher value than a thin ceramic of identical material and finish). When using this parameter to its full potential, it becomes necessary to create a different parameter-set for each material thickness.

```
models/interactions/radiation_pressure/verif/SIM_3_ORBIT/Modified_data/radiation_surface.py
 92 params = trick.RadiationParams()
...
 97 params.thermal.emissivity = 0.5
 98 params.thermal.heat_capacity_per_area = 50.0
 99
100 radiation.surface_factory.add_facet_params(params)
```

Note – Thermal facets can be used as regular facets for creation of a second, non-thermal interaction surface. So a surface could be created using *FlatPlateThermal* facets, with the thermal facet-parameters set. Then, a set of thermal material-parameters set for the radiation-pressure model, and a set of non-thermal material parameters of the same name set for the Aerodynamics model. Then the Radiation Pressure model would get a thermally active Radiation Surface, and the Aerodynamics model would get a thermally-inactive Aerodynamic Surface.

G.5 Collecting Forces and Torques [Aside]

We are about to study the environmental forces and torques that can be applied to a vehicle. These must be collected and applied to the vehicle, as outlined in section B.5.2.c.

G.6 Aerodynamic Drag

Document reference: *models/interactions/aerodynamics/docs/aerodynamics.pdf*

The Aerodynamic Drag Model calculates forces and torques resulting from incident and emitted radiation. There are two implementations for aerodynamic drag:

1. The simple implementation assumes that the aerodynamic cross-section is isotropic. It requires no surface definition.

2. The surface-based implementation provides an aerodynamic cross-section that is orientation-dependent. It requires the implementation of a an AerodynamicSurface, created from the Surface Model.

The implementation of the Aerodynamic Drag Model is typically found in one of three locations:

- in a simulation-object dedicated to aerodynamic drag,
- in a simulation-object dedicated to interactions in general, or
- in the simulation-object used to define the vehicle.

Regardless of its location, the implementation starts with the instantiation of an AerodynamicDrag object.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/S_define
 40    jeod::AerodynamicDrag aero_drag;
```

There are no initialization jobs necessary, and the run-time operation is limited to a single call:

```
57       (DYN, "scheduled") aero_drag.aero_drag(inertial_vel,
58                                              &atmos_state,
59                                              T_inertial_struct,
60                                              mass,
61                                              center_grav);
```

Note that this method requires significant data input, all of which are typically found in the vehicle simulation-object.

1. The inertial velocity is typically *vehicle.dyn_body.composite_body.state.trans.velocity*
2. The atmosphere-state is typically *vehicle.atmos_state*
3. The transformation matrix from inertial to structural frames is typically *vehicle.dyn_body.structure.state.rot.T_parent_this*
4. The mass is typically *vehicle.dyn_body.mass.composite_properties.mass*
5. The position of the center-of-mass expressed in the structural reference frame is typically *vehicle.dyn_body.mass.composite_properties.position*

In the case that this Aerodynamics Model is NOT implemented in the vehicle simulation object, the most straightforward way to access these variables is to create a *VehicleSimObject* reference in this simulation-object, populate it at construction time, and access the values directly. See section A.4.4 for details on this process.

```
Example
(DYN, "scheduled") aero_drag.aero_drag(veh.dyn_body.composite_body.state.trans.velocity,
                                       ... etc);
```

Common to both the simple and the surface-model implementation is the ability to turn the model on or off. The default is to be on.

models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/input_common.py

```
  6 aero_test.aero_drag.active    = True
```

G.6.1 Simple Aerodynamic Drag

This is the default implementation. It can (and probably should) be explicitly stated with the *use_default_behavior* flag.

models/interactions/aerodynamics/verif/SIM_VER_DRAG/SET_test/RUN_aero_drag/input.py

```
 12 aero_test.aero_drag.use_default_behavior = True
```

This simple implementation does not require (nor use even if one exists) the definition of a Surface Model surface. Instead, the cross-sectional characteristics are provided by specifying either the ballistic coefficient or a combination of the coefficient of drag and cross-sectional area. Alternatively, a constant drag force magnitude

can be provided; the drag force will then have the specified magnitude and be aligned with relative wind velocity.

G.6.1.a Ballistic Coefficient

Set the identifying option, and the ballistic coefficient.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/SET_test/RUN_aero_drag_BC/input.py
 13 aero_test.aero_drag.ballistic_drag.option = trick.DefaultAero.DRAG_OPT_BC
 14 aero_test.aero_drag.ballistic_drag.BC = 0.005
```

G.6.1.b Coefficient of Drag

Set the identifying option, the coefficient of drag, and the cross-sectional area (default to square-meters unless specified otherwise).

```
Example
 13 aero_test.aero_drag.ballistic_drag.option = trick.DefaultAero.DRAG_OPT_CD
 14 aero_test.aero_drag.ballistic_drag.area = 100
 15 aero_test.aero_drag.ballistic_drag.Cd = 2
```

G.6.1.c Constant Force Magnitude

Set the identifying option, and the force magnitude (default to Newtons).

```
Example
aero_test.aero_drag.ballistic_drag.option = trick.DefaultAero.DRAG_OPT_CONST
aero_test.aero_drag.ballistic_drag.drag = 100.0
```

G.6.2 Surface-based Aerodynamic Drag

This option is selected by setting the *use_default_behavior* flag to false.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/input_common.py
 18 aero_test.aero_drag.use_default_behavior = False
```

This implementation requires that a Surface Model surface has been instantiated and defined, along with the *AeroSurface* implementation of an Interaction Surface, and the *AeroSurfaceFactory* for generating the *AeroSurface* from the *SurfaceModel* implementation. (see section G.3 for details on the relation between the Surface, the Interaction Surface, and the Surface Factory).

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/S_define
 44    jeod::SurfaceModel surface;
 45    jeod::AeroSurface aero_surface;
 46    jeod::AeroSurfaceFactory surface_factory;
```

Note – it is not necessary to implement the Surface Model in this simulation-object, although it is typical to include the *AerodynamicSurface* and *AerodynamicSurfaceFactory* in here.

Once defined, the Aerodynamics Model must be able to access the Surface Model, this is done by calling the *set_aero_surface* method:

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/input_common.py
 19 aero_test.aero_drag.set_aero_surface(aero_test.aero_surface)
```

Remember, if the surface is defined in its own Modified-data file, that file must be included from the input file.

Note that this method really only sets the pointer. It is not necessary to have the AerodynamicSurface created at this point. Typically the AerodynamicSurface is created as an initialization-class job, called from the S_define; it is acceptable to call the *set_aero_surface* job from the input file beforehand.

There are four options for calculating the drag on each facet; these are set in the parameters list, so are applied to all facets utilizing the same material properties data-set. All options are based on the physics of free-molecular flow with the magnitude of the force resulting from the dynamic pressure ($\frac{1}{2}\rho v^2$), and the appropriate drag coefficients. These options are set as a parameter set in the surface implementation, so are assigned to a particular “type” of facet. Common to all options, it is necessary to instantiate a new set of aerodynamic facet parameters, associate this set of parameters with a facet type, and add it to the factory's parameter list.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/shuttle_plate_aero_drag.py
 98 params = trick.FlatPlateAeroParams()
 99 params.thisown = 0
...
107 params.set_name(“flat_plate_material")
...
109 aero_test.surface_factory.add_facet_params(params)
```

G.6.2.a Specular Interaction

This model works under the assumption that the molecules in the free-stream bounce off the surface with no thermal interaction, and do so with no transfer of momentum tangential to the plate. The net force is therefore normal to the plate.

It is selected by default, or ca be explicitly selected by setting the identifying coefficient method. This is most closely a material property, so is found in the material parameter set.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/shuttle_plate_aero_drag.py
106 params.coef_method = trick.AeroDragEnum.Specular
```

The drag coefficient may either be set to a constant value (default)

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/one_plate_accel_calc_coef_eps00.py
 64 params.drag_coef_spec = SPEC_DRAG_COEF
...
 69 params.calculate_drag_coef = False
```

or calculated at each pass.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/one_plate_accel_diff_max_coef.py
 71 params.calculate_drag_coef = True
```

If the drag coefficient is to be calculated at each pass, it is necessary to specify the free-stream temperature (*temp_free_stream*, typically around 1000K – 1500K for earth-orbit), and the gas constant (gas_const, for dry air, this has a value of $287\ J\ kg^{-1}\ K^{-1}$, default units are $J\ kg^{-1}\ K^{-1}$). This can be specified in the modified-data, or included as default data; both examples are given below:

models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/shuttle_aero_model.py

```
  4    aero_drag.param.gas_const = 287
  5    aero_drag.param.temp_free_stream = 1487
```

```
##include "interactions/aerodynamics/data/include/aero_model.hh"
...
jeod::AerodynamicDrag_aero_model_default_data    aero_drag_default_data;
...
("default_data") aero_drag_default_data.initialize( &aero_drag);
```

See the model documentation for details of the calculations.

G.6.2.b Diffuse Interaction

This model works under the assumption that the molecules in the free-stream are momentarily absorbed onto the surface, and then slide off carrying no momentum. The net force is therefore parallel to the relative velocity vector of the free-stream.

It is selected by setting the identifying coefficient method,

models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/one_plate_accel_diff_max_coef.py

```
 70 params.coef_method = trick.AeroDragEnum.Diffuse
```

then the drag coefficient may either be set to a constant value (default)

models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/one_plate_accel_calc_coef_eps00.py

```
 66 params.drag_coef_diff = DIFF_DRAG_COEF
...
 69 params.calculate_drag_coef = False
```

or calculated at each pass as for the Specular Interaction case (remembering to set the *temp_free_stream* and *gas_constant* values).

See the model documentation for details of the calculations.

G.6.2.c Mixed Interaction

This model works under the assumption that some fraction (*params.epsilon)* of the molecules in the free-stream interact specularly, and the balance interact diffusely

It is selected by setting the identifying coefficient method,

```
Example
params.coef_method = trick.AeroDragEnum.Mixed
```

and the relative weighting of the specular component (0 to 1)

```
Example
params.epsilon = 0.5
```

then the drag coefficients may either be set to constant values (specifying both the *drag_coef_spec* and *drag_coef_diff* values) or calculated at each pass, as for the Specular Interaction and Diffuse Interaction cases.

G.6.2.d Normal and Tangential Coefficients

With this method, drag coefficients normal and tangential to the facet surface (*drag_coeff_norm* and *drag_coeff_tang* respectively) are calculated at every pass. Setting the value *calculate_drag_coef* to false is an invalid option; doing so will result in an automatic reversal to true, with an accompanying warning. See the model documentation for details of the calculations with this option.

```
models/interactions/aerodynamics/verif/SIM_VER_DRAG/Modified_data/one_plate_torque.py
 69 params.coef_method = trick.AeroDragEnum.Calc_coef
```

G.6.3 Logging Aerodynamic Drag Data

The net aerodynamic drag force and torque are the most commonly logged data elements.

```
Example
for ii in range(0,3) :
  dr_group.add_variable("interactions.aero_drag.aero_force[" + str(ii) + "]")
  dr_group.add_variable("interactions.aero_drag.aero_torque[" + str(ii) + "]")
```

Exercise 27. Add Aerodynamic Drag

G.7 Radiation Pressure

Document reference:

models/interactions/radiation_pressure/docs/radiation_pressure.pdf

G.7.1 Introduction and Implementation

The Radiation Pressure calculates forces, torques, and temperature variations, resulting from incident and emitted radiation. There are two implementations for radiation pressure:

1. The simple implementation assumes that the radiation surface is spherical and isothermal. It produces no torques, and no forces due to emitted radiation. It requires no facet-by-facet surface definition but uses a single *RadiationDefaultSurface* instantiation instead.

2. The surface-based implementation provides a multi-faceted radiation surface, with each facet having its own orientation relative to the environmental radiation field. It requires the implementation of a *RadiationSurface*, created from the Surface Model.

The implementation of the Radiation Pressure Model is typically found in one of three places:

- in a simulation-object dedicated to radiation pressure,
- in a simulation-object dedicated to interactions in general, or
- in the simulation-object used to define the vehicle.
- Regardless of location, the implementation starts with the instantiation of an *RadiationPressure* object.

```
models/interactions/radiation_pressure/verif/S_modules/radiation.sm
  9 ##include "interactions/radiation_pressure/include/radiation_pressure.hh"
...
 22     jeod::RadiationPressure       rad_pressure;
```

There is one initialization job necessary; this call depends on which implementation is being used and is shown in the subsequent sections.

There is one run-time operation, common to both implementations:

```
 53       (DYNAMICS, "scheduled") rad_pressure.update(
 54            veh.structure,
 55            veh.mass.composite_properties.position,
 56            dyn_time.scale_factor,
 57            dyn_time.seconds);
```

Note the argument list; this method requires input from the time simulation-object and from the vehicle simulation-object. See section A.4.4 for details on passing data into this simulation-object from others.

Aside – For those who may be curious, these external references are required because of the way the Radiation Pressure model integrates the facet temperatures. With recent advances in integration modeling (see section I.4.2), there is now the concept of an improved interface, but it has not yet been integrated into the JEOD release.

The Radiation Pressure Model actually performs two related calculations – the forces and torques resulting from radiative interactions, and the corresponding temperature variability. Either or both may be on or off at any point in the simulation. Common to both implementations are the following three flags:

1. *active* provides the ability to turn the model off entirely. The default is to be on.
2. *calculate_forces* indicates whether forces and torques should be calculated
3. *thermal_active* indicates whether the facet/surface temperature is allowed to vary. Note – even for the simple isothermal surface, the temperature may still have a temporal variation.

Example
```
interactions.radiation_pressure.active = False
```

G.7.2 Simple Radiation Pressure Model

To use this implementation, a *RadiationDefaultSurface* must be instantiated:

models/interactions/radiation_pressure/verif/S_modules/radiation_simple_no_decl.sm
```
  4 ##include "interactions/radiation_pressure/include/radiation_default_surface.hh"
...
 10     jeod::RadiationDefaultSurface  rad_surface;
```

and the Radiation Pressure Model initialized with this surface:

```
 27       P_ENV  ("initialization") rad_pressure.initialize(
 28            dyn_mngr,
 29           &rad_surface);
```

Note – in this case, the initialize method takes two arguments.

This surface has the following values that must be specified:

Area

Specify either cross sectional area (*cx_area*, πr^2) or spherical surface area (*surface_area*, $4\pi r^2$).

interactions/radiation_pressure/verif/SIM_4_DEFAULT/SET_test/RUN_compare/input.py
```
 56 radiation_simple_inside.rad_surface.cx_area = 3.14
```

Susceptibility to radiation field

Specify either the radiation coefficient (*rad_coeff*), or a combination of the albedo (*albedo*, this is the fraction of incident radiation that is reflected) and the diffuse-parameter (*diffuse_parameter*, this is the fraction of the reflected light that is reflected diffusely – as opposed to reflected specularly)

```
 57 radiation_simple_inside.rad_surface.albedo = 0.5
 58 radiation_simple_inside.rad_surface.diffuse = 0.5
```

Note – for a spherical surface, $rad\_coeff = 1 + \frac{4}{9}(albedo)(diffuse)$.

Temperature

```
59 radiation_simple_inside.rad_surface.temperature = 270.0
```

Emissivity

The value *thermal.emissivity* indicates the ability to radiate energy away ($P = \epsilon \sigma A T^4$, where ϵ is the emissivity)

```
62 radiation_simple_inside.rad_surface.thermal.emissivity = 0.5
```

Temperature Response

Indicate whether the temperature is variable

```
60 radiation_simple_inside.rad_surface.thermal.active = True
```

If it is, specify the heat capacity for the entire vehicle.

```
63 radiation_simple_inside.rad_surface.thermal.heat_capacity = 628.0
```

Also, if the temperature is variable, optionally specify the rate at which thermal energy is added to the vehicle surface from vehicle internal processes, such as internal power generation. If not specified, this is assumed to be zero.

```
61 radiation_simple_inside.rad_surface.thermal.thermal_power_dump = 0.0
```

In order for the thermal power dump to be added (or subtracted) from the facet, the *include_internal_thermal_effects* flag must be set (in this example, that is not necessary because the value is zero anyway).

```
Example
interactions.radiation_pressure.thermal.include_internal_thermal_effects = True
```

G.7.3 RadiationSurface Implementation of Radiation Pressure

This implementation requires that a Surface Model surface has been instantiated and defined, along with the *RadiationSurface* implementation of an Interaction Surface, and the *RadiationSurfaceFactory* for generating the *RadiationSurface* from the *SurfaceModel* implementation. (see section G.3 for details on the relation between the Surface, the Interaction Surface, and the Surface Factory, and for details of defining the general surface).

```
models/interactions/radiation_pressure/verif/S_modules/radiation.sm
 10 ##include "interactions/radiation_pressure/include/radiation_surface_factory.hh"
 11 ##include "interactions/radiation_pressure/include/radiation_surface.hh"
 12 ##include "utils/surface_model/include/surface_model.hh"
...
 15 ##include "interactions/radiation_pressure/include/radiation_third_body.hh"
 16 ##include "utils/surface_model/include/flat_plate_thermal.hh"
 17 ##include "interactions/radiation_pressure/include/radiation_params.hh"
...
 23     jeod::RadiationSurfaceFactory surface_factory;
 24     jeod::RadiationSurface        rad_surface;
 25     jeod::SurfaceModel            surface;
```

Remember, if the surface is defined in its own Modified-data file, that file must be included from the input file.

The surface factory must first create the Radiation Surface:

```
models/interactions/radiation_pressure/verif/S_modules/radiation.sm
 43       P_ENV  ("initialization") surface_factory.create_surface(
 44           &surface,
 45           &rad_surface);
```

The initialization of the Radiation Pressure Model is performed using the method appropriate for a *RadiationSurface*; this links the Radiation Pressure Model to its Radiation Surface:

```
models/interactions/radiation_pressure/verif/S_modules/radiation.sm
 46       P_ENV  ("initialization") rad_pressure.initialize(
 47            dyn_mngr,
 48           &rad_surface,
 49            veh.mass.composite_properties.position);
```

Note that this method takes three arguments, whereas the simple model took only two. The difference is in the necessity for the location of the center of mass; since the simple model can produce no torque, this value is not needed.

G.7.3.a Defining the Surface Facets

When defining the surface model for the purposes of creating a Radiation Surface, it is necessary to use Thermal Facets (e.g. *FlatPlateThermal*, not *FlatPlate*). See the Thermal Rider Model (section G.4) for details on making facets thermally capable.

The parameters to be specified for each facet include

1. The general parameters for a generic facet – *position*, *normal*, *area*, *temperature*, and *param_name* (see section G.3.2)

2. The optional additional general parameters for a thermal facet – *thermal.active*, and *thermal.thermal_power_dump* (see section G.4.2).

3. The materials properties specific to the Radiation Interaction:

   (a) *albedo* is the fraction of incident radiation that is reflected

(b) *diffuse* is the fraction of the reflected radiation that is reflected diffusely (see the model documentation for an analysis of specular versus diffuse reflection).

Note – unlike the default spherical surface, there is no option for specifying a radiation coefficient. Radiation coefficients are typically empirically measured values based on a completed physical model of the vehicle; they are highly geometry-dependent and cannot, in general, be created analytically from values associated with individual surface elements.

(c) *thermal.emissivity* indicates the ability to radiate energy away ($P = \epsilon \sigma A T^4$, where ϵ is the emissivity)

(d) *thermal.heat_capacity_per_area* is the heat capacity per unit area of surface. See section G.4.2 for more details on these last two parameters).

models/interactions/radiation_pressure/verif/SIM_2_SHADOW_CALC/Modified_data/radiation_surface_a.py

```
 10 fpt.position  = [ 2.0 , 0.0, 0.0 ]
 11 fpt.normal  = [ 1.0 , 0.0 , 0.0 ]
 12 fpt.area  = 60.0
 13 fpt.temperature  = 500.0
 14 fpt.param_name = "radiation_test_material"
 15 fpt.thermal.active = True
 16 fpt.thermal.thermal_power_dump = 0.0
...
 95 params.set_name("radiation_test_material")
 96 params.albedo = 0.5
 97 params.diffuse = 0.5
 98 params.thermal.emissivity = 0.5
 99 params.thermal.heat_capacity_per_area = 50.0
```

Remember to ensure that the appropriate header files have been included in the S_define to allow the creation of facets at the Modified-data level. This applies to both the facet and parameter headers:

models/interactions/radiation_pressure/verif/S_modules/radiation.sm

```
 16 ##include "utils/surface_model/include/flat_plate_thermal.hh"
 17 ##include "interactions/radiation_pressure/include/radiation_params.hh"
```

Then the parameters can be set in modified-data:

models/interactions/radiation_pressure/verif/SIM_3_ORBIT/Modified_data/radiation_surface.py

```
 92 params = trick.RadiationParams()
 93 params.thisown = 0
 94 params.name =  "radiation_test_material"
 95 params.albedo = 0.5
 96 params.diffuse = 0.5
 97 params.thermal.emissivity = 0.5
 98 params.thermal.heat_capacity_per_area = 50.0
 99
100 radiation.surface_factory.add_facet_params(params)
```

G.7.4 Setting the Radiation Field

The illumination of each facet depends on several factors:

1. The strength of the radiation field. This is affected by:

   (a) The source of the radiation

(b) The distance to the source

(c) The shadowing effects of any intervening bodies.

2. The relative orientation of the facet to the radiation field.

G.7.4.a Illuminating Bodies

The radiation source defaults to be Sun. There is also an albedo radiation model available in the contributions. It has not been thoroughly verified so does not appear in the regular JEOD release, but this can also provide additional radiation sources.

G.7.4.b Shadowing Bodies

The intervening bodies are referred to as third-bodies in the model, and are represented by the class *RadiationThirdBody.* There are two options for calculating the shadow cast by third-bodies:

1. Cylindrical – this is the easier calculation. It blocks out the cross-section of the third-body uniformly to an infinte distance

2. Conical – this is the more realistic option. It considers both the source and third-body to have a finite size, and considers the lines from the limb of one to the limb of the other. Behind ("downstream" in the radiation field) the third-body there is a converging cone of totality (total radiance blockage) and a diverging conical frustrum of partial blockage.

To include shadowing effects, be sure that the appropriate header file is included in the S_define. Instances of *RadiationThirdBody* are created at the input / Modified-data file level

models/interactions/radiation_pressure/verif/S_modules/radiation.sm

```
 15 ##include "interactions/radiation_pressure/include/radiation_third_body.hh"
```

Each third-body is instantiated and named (the name must match with one of the planetary bodies already registered with the Dynamics Manager). Each third-body is individually activated or de-activated, and each also identifies the desired type of shadow to be cast. Each is added to the Radiation Pressure Model as it is defined.

models/interactions/radiation_pressure/verif/SIM_3_ORBIT/Modified_data/third_bodies.py

```
  7 third_body = trick.RadiationThirdBody()
  8 third_body.thisown = 0
  9 third_body.set_name("Earth")
 10 third_body.active = True
 11 third_body.shadow_geometry = trick.RadiationThirdBody.Conical
 12
 13 radiation.rad_pressure.add_third_body(third_body)
```

G.7.5 Logging the Radiation Pressure Model

The net radiation-pressure force and torque are the most commonly logged data elements. Secondary data includes the flux magnitude.

```
Example
for ii in range(0,3) :
  dr_group.add_variable("interactions.rad_pressure.force[" + str(ii) + "]")
  dr_group.add_variable("interactions.rad_pressure.torque[" + str(ii) + "]")
```

During the verification process, it was necessary to develop additional methods for monitoring the facets individually. These data can still be accessed by adding the necessary methods from the verification directory:

```
models/interactions/radiation_pressure/verif/S_modules/radiation.sm
 13 ##include "interactions/radiation_pressure/verif/include/radiation_data_recorder.hh"
...
 26     jeod::RadiationDataRecorder   data_rec;
...
 58       (DYNAMICS, "scheduled") data_rec.record_data (
 59           &rad_surface);
```

The individual facet data can then be accessed from the *data_rec* instance.

```
Example
for ii in range(0,6) :
  dr_group.add_variable("radiation.data_rec.temperature[" + str(ii) + "]" )
  dr_group.add_variable("interactions.rad_data_rec.power_absorb[" + str(ii) + "]" )
  for jj in range(0,3) :
    dr_group.add_variable("interactions.rad_data_rec.force[" + str(ii)+"]["+str(jj))+ "]")
```

Note – these methods were designed for use in a particular case. They have not been verified to the same level as the rest of JEOD. Use with caution.

Exercise 28. Add Radiation Pressure

G.8 Contact

Document reference: ***models/interactions/contact/docs/contact.pdf***

G.8.1 Introduction

The Contact Model provides the ability to simulate very-close proximity operations, including collision, docking, grappling, and berthing. It provides forces and torques encountered during contact events.

While the radiation pressure and aerodynamic models modeled the interaction of a facet with some environmental parameter, the contact model considers interactions of a facet with another facet. This leads to some extra complexity in handling the possible pairings of facets. The extra complexity is managed within the Contact Model, but that means that all facet pairings have to be registered with the model. More on this later.

G.8.2 Defining the Surface

G.8.2.a Surface

The Surface Model implementation for each vehicle must be associated with that vehicle. The value *struct_body_name* must be set to the name of the vehicle.

```
Example
vehicle.surface.struct_body_name = “vehicle”
```

G.8.2.b Facets

To model the aerodynamic and radiation-pressure effects on a vehicle, it is typically sufficient to model the surface as a closed surface comprising multiple flat facets. The actual geometry of those facets not required, only the effective frontal area. That is not true for vehicle-vehicle contact; testing for the co-location of two facets requires that we know their geometry. Conversely, in most applications, the contacting facets are known ahead of time, so the Contact Model needs only definition of a subset of the whole surface. Consequently, two new facet-types have been defined for use in the Contact Model:

1. *FlatPlateCircular* is a simple extension of the *FlatPlate* to include a *radius* and represent a circular geometry. The *position* of this facet represents the position of the center of the facet, the *normal* is perpendicular to the facet.

2. *Cylinder* is a simple extension of *CircularFlatPlate* to include a *length*. This facet type represents a right-circular cylinder. The *position* of this facet represents the position of the centroid of the cylinder, the *normal* is perpendicular to one of the circular ends.

A Surface Model intended for the Contact Model should be constructed exclusively from these facet types. Note that for the Contact Model, the surface does not need to be closed. For a docking procedure, for example, it is sufficient to model the entire vehicle surface as a single circular flat plate.

Aside: Note that the surface definition for the Contact Model surface is often incompatible with that for the Radiation Pressure and Aerodynamic models; it is necessary to provide two surface models. Although the ability to model aerodynamics and radiation pressure from the circular flat plate would be an easy extension of the existing code, it is not possible to build a closed surface from flat circles. Thus, while the Radiation Pressure and Aerodynamics models could, in principle, interpret a circular flat plates as they would any other flat plate of undefined geometry, the surface so constructed cannot possibly represent a real surface. The radiation pressure and aerodynamics models do not have the functionality to model their respective interactions with a cylindrical facet. Recommended practice is to provide a complete, closed surface for radiation pressure and aerodynamics, and a second minimal surface representing only those facets of interest for the contact model.

When specifying a facet for use with the contact model, it is necessary to give two additional values that were not necessary for the aerodynamics or radiation pressure models:

1. *name* gives each facet a name, used for creating pairs of facets for interacting. The name of the facet is generally unconstrained, although names should be unique, and should not match the names of any *MassPoint* instances on the same vehicle (part of the initialization process involves the creation of a *MassPoint* instance called *name*).

2. *mass_body_name* is the name of the vehicle object to which the facet is associated. This name should match with the name of one of the *DynBody* instances already registered with the Dynamics Manager. **Important notes:**

    1. Even though it is called *mass_body_name*, the model needs a dynamic state, so the *MassBody* carrying that name **must** be within a *DynBody.*

    2. The name **must** match across all facets of a given surface. To associate facets independently with, say, a parent and child in an attached *DynBody* scenario **requires** two independent surface models. This is a good idea anyway; recall that the surface for the contact model need only define the utility areas (it does not need to be a complete closed surface), so there is no benefit gained from providing one all-encompassing surface during attached operations. From an operational efficiency perspective, each *DynBody* should have its own contact surface, whether it is attached or free-floating.

As with the other models, we also need to specify the *param_name* for the new interaction surfaces, although this has a slightly different application.

```
Example
fpc = trick.FlatPlateCircular()
fpc.thisown = 0
fpc.position  = trick.attach_units( "m",[ 4.0 , 2.0, 0.0 ])
fpc.normal  = [ 1.0 , 0.0 , 0.0 ]
fpc.radius  = trick.attach_units( "m",1.0)
fpc.param_name = "steel"
fpc.mass_body_name = "vehicle"
fpc.name = "docking port face"

vehicle.surface_model.add_facet(fpc)
```

G.8.2.c Contact Facet Parameters

When we defined the surface for aerodynamics and radiation pressure, we specified a set of facet parameters that modeled how the facet material responds to the particular environment. For the contact mode, the environment is another Contact Facet on another vehicle; the interaction between these two facets depends on the combined material properties of said facets.

Consequently, we only define the name of the material, and define the actual behavioral parameters for each material-material pair.

For example, suppose a facet on vehicle A is made of steel, and a facet of vehicle B made of ceramic. We define the parameter names for the two facets as they are added to the respective surfaces:

```
Example
fpc = trick.FlatPlateCircular()
...
fpc.param_name = "steel"
...
vehicleA.surface_model.add_facet(fpc)


fpc = trick.FlatPlateCircular()
...
fpc.param_name = "ceramic"
...
vehicleB.surface_model.add_facet(fpc)
```

Then we make instances of ContactParams to represent these materials, and simply add those to the facet-params list (as we did for the other interactions, but without specifying any data values)

```
Example
cp = trick.ContactParams()
cp.thisown = 0
cp.name = “steel”
vehicleA.contact_surface_factory.add_facet_params(cp)

cp = trick.ContactParams()
cp.thisown = 0
cp.name = “ceramic”
vehicleB.contact_surface_factory.add_facet_params(cp)
```

G.8.3 Defining the Pair Interactions

The only interaction possible in the Contact Model is between a pair of Contact Facets. It is necessary to consider two aspects of this interaction:

1. The general physics behind the model of how the facets will interact
2. The specific implementation of that physics model to the materials involved in each interaction.

G.8.3.a Selecting a Physics Model (pair-interaction method)

Spring-Damper-Friction (*SpringPairInteraction*)

One option is combination of a spring-damper with sliding friction system. Perpendicular to the plane of contact, $\vec{F}_p = -k\,\vec{x} - b\,(\vec{v}\cdot\hat{\vec{x}})\,\hat{x}$, with $\vec{x}$ representing the penetration vector, and $\vec{v}$ the relative velocity of the two facets.

Note – facets deform under contact; penetration in this sense refers to the penetration that the facet *would* have through the other if neither had deformed.

Additionally, the frictional force is given by $\vec{F}_f = \mu\,|\vec{F}_p|\,\hat{s}$ where $\hat{s}$ is the unit vector representing the projection of the velocity vector onto the contacting surface.

Note that any one of these three terms (spring, (k), damper (b), friction (μ)) can be eliminated bys etting the appropriate constant to zero.

Additional physics models may be implemented.

G.8.3.b Define the Interaction Pairs

To implement a pair-interaction method, make an instance of one, populate its data, and register it with the Contact Model instance. This must be done for each combination of materials that could be interacting.

The data for the *SpringPairInteraction* include:

- Names of the materials / parameters of each interacting surface
- spring constant between these two materials
- damping constant between these two materials
- coefficient of kinetic friction between these two materials.

Continuing with the previous example:

```
Example
pair = trick.SpringPairInteraction()
pair.thisown = 0
pair.params_1 = “steel”
pair.params_2 = “ceramic”
pair.spring_k = trick.attach_units( "N/m",10000.0)
pair.damping_b = trick.attach_units(“N*s/m”,200.0)
pair.mu = 0.1

interactions.contact.register_interaction(pair)
```

Note - the data provided for these interacting pairs are usually empirically derived. The values presented in this example are purely illustrative, and are not in any way intended to represent the data for any particular vehicle.

G.8.4 Setting up the S_define

G.8.4.a Instantiations

A simulation including the Contact Model requires the following:

- at least two vehicles (aka *DynBody* instances), and for each such vehicle:
  - *SurfaceModel*, the contact-model compatible surface model for each vehicle.
  - *ContactSurface*, the contact-model interaction surface, created from the surface model for each vehicle.

- *ContactSurfaceFactory*, the tool by which the surface is transformed into the contact surface.
- *Contact*, the actual contact model instance.

As with the other interactions, the *SurfaceModel* instance typically stays with the vehicle simulation-object. The *ContactSurface* and *ContactSurfaceFactory* typically stay together, typically in the vehicle simulation-object, or in a separate *interactions* simulation-object, or in a simulation-object dedicated to the Contact Model. Having redundant instances of the *ContactSurfaceFactory* is not in any way detrimental to the simulation, so each vehicle could have its own. Conversely, there should be only one contact model, so it is typically found separate from the vehicle in either an *interactions* simulation-object, or in a simulation-object dedicated to the Contact Model.

Example (pseudo-code)

```
class VehicleSimObject: public Trick::SimObject {
...
##include "utils/surface_model/include/surface_model.hh"
##include "utils/surface_model/include/flat_plate_circular.hh"
...
   jeod::SurfaceModel            surface_model;
...
}
VehicleSimObject vehicleA(dynamics.dyn_manager);
VehicleSimObject vehicleB(dynamics.dyn_manager);
```

Example (pseudo-code, continued)

```
class InteractionSimObject: public Trick::SimObject {
...
##include "interactions/contact/include/contact_surface.hh"
##include "interactions/contact/include/contact_surface_factory.hh"
##include "interactions/contact/include/contact_params.hh"
##include "interactions/contact/include/spring_pair_interaction.hh"
##include "interactions/contact/include/contact.hh"
...
    jeod::ContactSurface          vehA_contact_surface;
    jeod::ContactSurface          vehB_contact_surface;
    jeod::ContactSurfaceFactory   contact_surface_factory;
    jeod::Contact                 contact;
...
}
InteractionSimObject interactions(...)
```

G.8.4.b Initialization

Initialization is a 3-part process:

1. Associate the surface model with the vehicle (*initialize_mass_connections* method). This connection is based on the *mass_body_name* set in each facet.

```
Example
class VehicleSimObject: public Trick::SimObject {
...
P_BODY ("initialization") surface_model.initialize_mass_connections( dyn_manager );
...
}
```

2. Create the Contact Surfaces (*create_surface* method)

```
Example
class InteractionSimObject: public Trick::SimObject {
...

...
InteractionSimObject(DynManager & dyn_manager_in,
                     SurfaceModel & vehA_surface_in,
                     SurfaceModel & vehB_surface_in)
:
dyn_manager(dyn_manager_in),
vehA_surface(vehA_surface_in),
vehB_surface(vehB_surface_in),
{
P_BODY ("initialization") contact_surface_factory.create_surface(
                                                & vehA_surface,
                                                & vehA_contact_surface);
P_BODY ("initialization") contact_surface_factory.create_surface(
                                                & vehB_surface,
                                                & vehB_contact_surface);
```

Note – the vehicle surface-model instances are passed into this simulation-object to provide input to the *create_surface* method. The Dynamic Manager instance is passed in to provide input to the next step.

3. Register the facet pairs with the Contact Model (*register_contact* method) and initialize the model. There are five methods for registering facets, and some care should be taken in selecting the most appropriate method, particularly if the contact surfaces are complex. To identify whether there is contact between a pair of contact-facets, it is necessary to know the relative states of every pair of facets (or at least, every pair of interest). The process of calculating this relative state is time-consuming, and with a large number of facets in the simulation, checking every possible facet-facet pairing (with *n* facets, there are *n(n-1)* possible pairs) will quickly become prohibitive. With large *n*, care must be taken when registering contact facets with the Contact Manager so as to avoid unnecessary pairings.

   Note that pairings of facets that are in the same mass tree (i.e. on the same composite vehicle) will not be tested for contact. If registered as a pair, they will be added to the list of pairs, but at each processing call, every pair is tested for being on the same tree, and skipped over if the test fails. Realize then, that since every *DynBody should* have its own surface model for contact purposes, the applications where it would be desirable to register a pair of facets from the same contact surface are very few and far between.

   The five methods follow:

   (a) One surface at a time. Input arguments are the array of contact facets in the contact surface, and the number of contact facets in the contact surface. This process will produce a pairing of every contact facet in this contact surface with every other contact facet registered at the time that the *initialize_contact* method is called, including those on the same surface. This is the easiest method to implement, and the slowest method to execute because of the *n(n-1)* geometric expansion in the number of available pairs (mentioned above). This method is rarely useful, primarily because of the

usually-unnecessary pairing of facets on the same surface. The calling sequence MUST be *register_contact* followed by *initialize_contact.*

```
Example
P_BODY  ("initialization") contact.register_contact(
                                                vehA_contact_surface.contact_facets,
                                                vehA_contact_surface.facets_size);
P_BODY  ("initialization") contact.register_contact(
                                                vehB_contact_surface.contact_facets,
                                                vehB_contact_surface.facets_size);
P_DYN  ("initialization") contact.initialize_contact(&dyn_manager);
```

(b) One facet at a time. The single arguments provides a pointer to the specific contact facet. This process will produce a pairing of this facet with every other contact facet registered at the time that the *initialize_contact* method is called, including those on the same surface. This can greatly cut down on redundant pairings, particularly if the contact surface is unnecessarily complex, but is more cumbersome to implement. The scenarios in which this method is optimal are limited. The calling sequence MUST be *register_contact* followed by *initialize_contact.*

```
Example
P_BODY  ("initialization") contact.register_contact(
                                            vehA_contact_surface.contact_facets[1]);
P_BODY  ("initialization") contact.register_contact(
                                            vehB_contact_surface.contact_facets[0]);
P_BODY  ("initialization") contact.register_contact(
                                            vehB_contact_surface.contact_facets[2]);
P_DYN  ("initialization") contact.initialize_contact(&dyn_manager);
```

This example will produce 3 contact-pairs – facet #1 on surface A with facet #0 on surface B

facet #1 on surface A with facet #2 on surface B

facet #0 on surface B with facet #2 on surface B

(c) All contact facets on one contact surface with all contact facets on another. The four arguments necessary provide the array of contact facets and number of contact facets on each surface. This method avoids the redundancy of pairing facets with facets on the same surface. It is typically the most useful method when there are more than one or two contact facets on a surface. The calling sequence MUST be *initialize_contact* followed by *register_contact.*

```
Example
P_DYN  ("initialization") contact.initialize_contact(&dyn_manager);
P_DYN  ("initialization") contact.register_contact(
                                                vehA_contact_surface.contact_facets,
                                                vehA_contact_surface.facets_size,
                                                vehB_contact_surface.contact_facets,
                                                vehB_contact_surface.facets_size);
```

Note that the priority on the register_contact method has shifted from *P_BODY* to *P_DYN*. This method includes the initialization of the relative dynamic state between the facets.

(d) Any specific pairing of one facet with another. The two arguments necessary are pointers to the two facets. This method is the most efficient at runtime because only those pairings that are specifically useful are created. It is typically the most useful method when there are only one or two contact

facets on each surface. However, it can be very cumbersome to set up when there are many potential pairings. The calling sequence MUST be *initialize_contact* followed by *register_contact.*

```
Example
P_DYN  ("initialization") contact.initialize_contact(&dyn_manager);
P_DYN  ("initialization") contact.register_contact(
                                                  vehA_contact_surface.contact_facets[1],
                                                  vehB_contact_surface.contact_facets[4]);
```

Note – contact_facets is an array of pointers to the facets, so de-referencing the array once (contact_facets[1]) leaves a pointer to the contact-facet, as needed.

(e) Any combination of the above. Note that the code protects against duplicating the same pairing.

Note also that in creating the pairs using any of the methods outlined above, there must be an interaction already defined between the two materials of the two facets (see section G.8.3.b for details on defining the material-material behavior) in order to create the associated *ContactPair* instance. If a group of facets is registered, then those pairs which have a defined process will be added as interaction-pairs, and those pairs without a process will be skipped over.

Example
Suppose that both contact surfaces had some facets made of “steel” and others made of “ceramic”, that the “steel-steel” and “steel-ceramic” interactions were defined, and that the facets were registered using method (c) above. The pairings of a “ceramic” facet with another “ceramic” facet will be skipped over because that interaction remains undefined. There will be no *ContactPair* instances created between those facets. If those facets were to come in contact, this implementation would not be able to recognize it.

G.8.4.c Running the Contact Model

There are two methods to be called at run-time. Because contact forces can be very large and typically change on very short timescales, these methods should both be run at the derivative rate (i.e., as fast as possible). If contact is to be carefully modeled, it may be advisable in some situations to slow the simulation down by manipulating the rate at which time advances when contact is detected (see section B.2.3.e for a discussion on time-advance rates).

1. The first method, *Contact::check_contact*, is called from the contact model itself. It processes all registered pairs, regardless of surface, verifying that both facets in the pair are active, in different mass-trees, and in range of a potential contact (this is based on the size of the facets and on the parameter *contact_limit_factor*, see below). Any pairs that pass these tests go through the relative-state and contact force computations.

2. These forces and torques so computed then need to be applied to the respective vehicles (which are surface-specific). Therefore, the second method, *ContactSurface::collect_forces_torques*, is called from each contact-surface to collect the forces and torques from each contact-facet in that contact-surface. Since each contact-surface is associated with only one vehicle, that collected value can then be applied to the vehicle using the conventional collect mechanism.

```
 Example
class InteractionSimObject: public Trick::SimObject {
...

("derivative")  contact.check_contact();
("derivative")  vehA_contact_surface.collect_forces_torques();
("derivative")  vehB_contact_surface.collect_forces_torques();
```

Note that method *check_contact* must be called before method *collect_forces_torques.* There should be one call to *check_contact* for each implementation of the Contact Model (typically one per simulation), and one call to *collect_forces_torques* for each contact-surface with a contact-facet registered in that Contact Model. Note also that the vehicle state should be updated before either of these are called; this would usually not be a concern. If the *ContactSurface* instance is in a different simulation-object to the *Contact* instance, be sure to apply appropriate priority levels to these function calls.

G.8.4.d Collecting the Contact Forces and Torques

Each *ContactSurface* has a *contact_force* and *contact_torque* 3-vector that are populated with the *collect_forces_torques* method.

```
Example
vcollect veh_A.dyn_body.collect.collect_environ_forc jeod::CollectForce::create {
   interactions.vehA_contact_surface.contact_force
};
vcollect veh_B.dyn_body.collect.collect_environ_forc jeod::CollectForce::create {
   interactions.vehB_contact_surface.contact_force
};

vcollect veh_A.dyn_body.collect.collect_environ_torq jeod::CollectForce::create {
   interactions.vehA_contact_surface.contact_torque
};
vcollect veh_B.dyn_body.collect.collect_environ_torq jeod::CollectForce::create {
   interactions.vehB_contact_surface.contact_torque
};
```

G.8.5 Setting the Input File

With the exception of defining the surface, which has been covered already, there is minimal control of the model available from the input file. The two features available allow the user to activate/deactive individual contact facets, and set the threshold separation distance between facet-pairs that will trigger whether to calculate the contact forces.

G.8.5.a Activating Facets

All contact_facet have an *active* flag that can be set directly, and defaults to *true.* Any facet-pair that has one (or both) facets inactive cannot produce a contact force. This is a good technique to speed a simulation along when contact is known to be impossible for specific facets.

```
Example
vehA_contact_surface.contact_facets[1]->active = False
```

This value can be set at any time after the contact facets have been created. It is not available during the regular assignments in the input file (which are read too early), but can be scheduled with a read statement, or set to respond to some dynamic event.

In addition, the Contact Model itself has an active flag that allows the entire model to be turned off when contact between all pairs is impossible (e.g. far-field operations).

```
Example
interactions.contact.active = False
```

G.8.5.b Setting Range Limits

Each *ContactPair* instance has a value *interaction_distance* that is accessed for testing whether the facets are close enough for a contact to be possible. This value should not be set directly – it is protected, and it would have to be accessed through the Contact Model's *contact_pair* vector of *ContactPair* instances, with no means of ensuring that the correct pair has been chosen. However, this value can be manipulated by setting the *contact_limit_factor* value in the Contact Model.

Each facet has an imaginary sphere of some determined radius around it, encompassing the entire facet. When a facet pair is created, the *interaction_distance is* calculated to be a multiple of the sum of the two sphere-radii from the two facets. That multiple is the *contact_limit_factor.* A value of 1.0 means that the calculations will only be performed if the two spheres intersect. A value of 2.0 means that the calculations will only be performed if the two facets are within two times the sum of the sphere-radii of each other.

The default value of 0.0 provides a special case, meaning that the calculations will always be performed.

```
Example
contact.contact_limit_factor = 1.0;
```

Note that this value is an initialization parameter only. Once the pairs are created, their *interaction_distance* cannot be manipulated again.

Exercise 29. Utilizing the Contact Model

G.9 Surface Articulation

Document reference: *models/utils/surface_model/docs/surface_mode.pdf*

G.9.1 Introduction

The articulation model allows facets within a surface to move with respect to one another. The resulting forces and torques associated with the motion of the surface components are not modeled, the main vehicle does not respond dynamically to the motion of the facets. This is simply a utility function to allow for the reconfiguring of the surface to model events such as the re-orientation of solar panels. Furthermore, all motions are discrete, instantaneous shifts, although continuous motion can be approximated with small time-steps.

The premise of the model is that all facets are associated with a *MassBody* (or derivative thereof, see section B.5 for an introduction to *MassBody and DynBody*). Each vehicle comprises one or more *MassBody* elements which are attached to each other with some relative position; that relative position (including orientation) may be changed (see section F.6 for information on building and manipulating compound vehicles). When a *MassBody* moves relative to another to which it is attached, it can take its facets with it. Hence, by associating facets with a *MassBody*, those facets move relative to one another. Facets may move in a group (multiple facets associated with the moving *MassBody*) or individually (single facet associated with the *MassBody*).

To allow this model to function, it is necessary to configure ALL facets in the surface for articulation by associating ALL facets in the surface with a *MassBody* in the same mass-tree. If ANY facet is not so configured, then NO facets can be articulated. Being configured for articulation does not mean that the facet has to move, just that it can move if commanded.

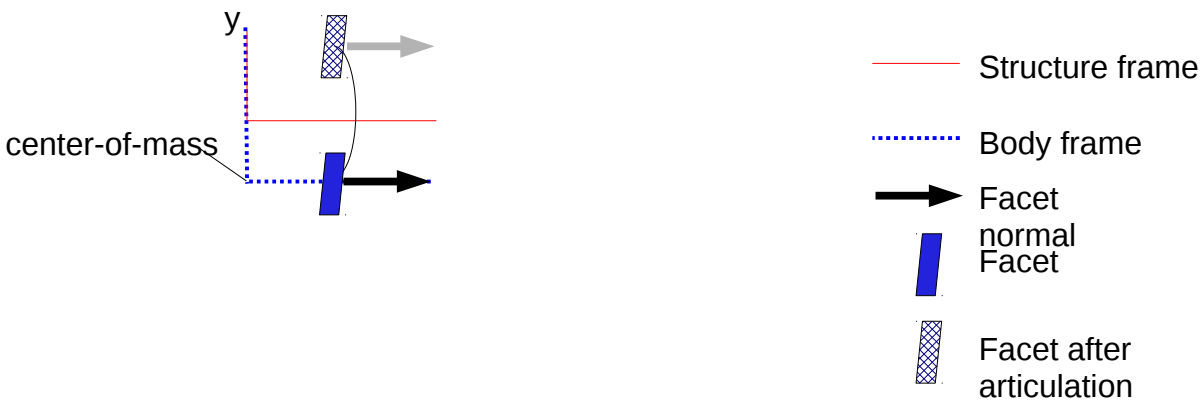
G.9.1.a WARNING

When moving one *MassBody* with respect to another, the *offset* and orientation values specified are typically between the structure-points (which provide the origins of the respective structural reference frames for *DynBody* objects). Particularly when the *MassBody* center-of-mass (origin of the body frame) differs from the structure-point, it is VERY IMPORTANT to remember that the specified rotations are with respect to the structure-point, and not the center-of-mass.

Example

Suppose a *DynBody* has structural and body frames aligned, but offset on the y-axis. Suppose further that a facet is centered on the x-axis of the body frame, with a normal along the x-axis, as shown in the illustration below.

The facet has a normal parallel to the x-axis of the structural frame, but it is NOT positioned on said axis. A specified articulation about the structural x-axis will maintain the orientation of this facet normal, but will move it in the y-z plane because of the y-axis offset from the STRUCTURAL origin. A user expecting the body to rotate about its center of mass, leaving this facet where it was, will obtain erroneous results.



G.9.2 Initializing the Articulation Model

G.9.2.a Facet Settings

The facets settings for the articulation model are relative to the structure-point of its specific *MassBody*. This is distinct from the facet settings for the Surface Model in general, which are relative to a common point (typically the structure-point of the root of the mass-tree) for all facets in the surface.

The following values are set:

- *mass_body_name*, the name of the *MassBody* with which this facet is to be associated
- *local_position*, the position of the facet with respect to its associated *MassBody*:
  - expressed in that *MassBody* structural frame
  - Defaults to [0,0,0]
- the orientation (if needed) of the facet with respect to its associated *MassBody*
  - Expressed in frame appropriate to the facet type
  - For a *FlatPlate*, this is set by the value *local_normal*, and is interpreted as being in the structural frame of its associated *MassBody*

```
Example
flat_plate = trick.FlatPlate()
flat_plate.thisown = 0
flat_plate.local_position = set_units("m", [2.0 , 0.0, 3.4 ])
flat_plate.area  = set_units("m2", 119.4454385 )
flat_plate.local_normal = [1.0 , 0.0 , 0.0  ]
flat_plate.mass_body_name = “vehicle solar-array #1”

vehicle.surface.add_facet(flat_plate)
```

Note that the *local_position* and *local_normal* are not in addition to *position* and *normal*, they are in place of those values. The values *position* and *normal* are computed relative to some master frame from the position and orientation of the facet relative to its *MassBody* and the position and orientation of that *MassBody* relative to the master frame.

G.9.3 Setting the S_define

G.9.3.a Initialization

Create the mass-tree by attaching the respective *MassBody* elements. Note that this should be performed with the attach processes provided by the Body Action Model. Non-recommended practices that may also work include calling the attach functionality directly from the S_define, or from the input file (subject to constraints). See section F.6 for options on attaching bodies.

The model is initialized with a call to the *SurfaceModel* method *initialize_mass_connections.*

```
(based on models/utils/surface_model/verif/SIM_ARTICULATION/S_define)
 81    ("initialization") surface.initialize_mass_connections( dyn_manager );
```

Notes:

1. The initialization may be performed before or after the masses are attached, but the attachments must be performed before the run-time calls are made. Recommended practice is to schedule this call with a *P_BODY* or lower priority, so that the Body Action attachments will be performed first.

2. While it is possible to build a mass tree without specifically registering *MassBody* instances with the Dynamics Manager, that is not true for running the *initialize_mass_connections* method. This will look for *MassBody* instances by name in the Dynamics Manager, so the Dynamics Manager must know of the respective masses, and they must be named. The method *add_mass_body* registers a *MassBody* with the Dynamics Manager. While *DynBody* instances must also be registered, any unregistered *DynBody* instances will be causing failures for other reasons (ALL *DynBody* instances MUST be registered).

```
Example
    P_ENV ("initialization") dyn_manager.add_mass_body(massA);
```

G.9.3.b Run-time

The articulation may be updated routinely or irregularly as needed. Note that the bodies are already attached, they just need to be moved. The recommended process for rearranging the *MassBody* elements is either a sequence of instances of the Body Action class *BodyReattach (*for irregular discrete motions), or a routine call to the *MassBody* method *reattach* (for regular, smooth motions).

In this example, we consider the routinely scheduled update. There are two components – the act of moving the *MassBody* elements around relative to one another, and the act of updating the surface to reflect the new arrangement.

See section F.6 for details on the attach options, including reattach.

```
(based on models/utils/surface_model/verif/SIM_ARTICULATION/S_define)
 86    (1.0, "environment") mass_bodyb.reattach(offset_b,
 87                                             T_pstr_cstr_b);
```

Note – the body attachment must have been previously applied in order to perform a reattach. Invalid calls to reattach will result in simulation termination.

The Surface Model method *update_articulation* calls the Facet method *update_articulation* on all of the facets in the Surface to update the facet position and orientation in the Surface Model.

```
(based on models/utils/surface_model/verif/SIM_ARTICULATION/S_define)
 93    (1.0, "environment") surface.update_articulation();
```

Note – if, at any time, the *update_articulation* method is run when there are facets in the surface associated with *MassBody* objects that are NOT in the same tree, **the simulation will immediately terminate**. There is no warning! Simulations involving detach processes mid-simulation that sever the mass-tree are particularly prone to such behavior. Care must be taken with such simulations to ensure that any articulating surface(s) does not encompass both pieces of the severed tree.

G.9.4 Setting the Input File

In addition to setting the values for each facet, there are two settings for the general Surface Model that may be found either with the facet specifications, or independently in the input file:

- *struct_body_name* sets the name of the master structural frame of the vehicle.
    - Position and orientation of facets in the Surface Model will be expressed with respect to this frame
    - This is typically (but not necessarily) the structural frame associated with the *MassBody* at the root of the mass tree.
- *articulation_active* is a boolean that indicates whether articulation is possible. The default value of this flag is *false*, so it must be deliberately set for articulation to function.

Notes:

1. This Articulation Model will redefine the position and orientation values of the facets with respect to the *struct_body_name* frame as the *MassBody* components move around with respect to this frame.

2. *mass_body_name* and *struct_body_name* must be legal names of MassBody objects registered in the Dynamics Manager

   (a) The *MassBody* objects must already be registered (with either *DynBody::initialize_model*, or *DynamicsManager::add_mass_body* before initializing the articulating surface.

3. *mass_body_name* and *struct_body_name* may be equal, but must both be specified.

4. All instances of *mass_body_name* must be in the same mass-tree as the *struct_body_name.*

```
Example
vehicle.surface.struct_body_name = "vehicle"
vehicle.surface.articulation_active = True
```

Exercise 30. Articulating Surfaces

Exercise 31. Smooth Rotation, with Radiation Pressure

G.10 External Effectors

JEOD does not model external effectors, but it can include their effects if they are otherwise implemented. With an instantiation of type *Force* and *Torque*, external forces and torques may be collected into the *effector_forc* and *effector_torq*, as outlined in section B.5.2.c.

```
verif/SIM_dyncomp_structure/S_define
 63 ##include "dynamics/dyn_body/include/force.hh"
 64 ##include "dynamics/dyn_body/include/torque.hh"
...
170    jeod::Force         force_extern;
171    jeod::Torque        torque_extern;
...
677 vcollect sv_dyn.body.collect.collect_effector_forc jeod::CollectForce::create {
678    sv_dyn.force_extern
...
685 vcollect sv_dyn.body.collect.collect_effector_torq jeod::CollectTorque::create {
686    sv_dyn.torque_extern
```

It is essential to recognize that these forces and torques – as collected in this example – will be applied to the *DynBody* identified as *sv_dyn.body.* The force is expressed in the structural reference frame of that *DynBody*. The torque is taken with a moment arm from the center of mass of the *DynBody*, and is also expressed in the structural reference frame.

Exercise 32. Simple Simulation of Vehicle with Thrusters

Chapter H Additional State Representations

Document reference: ***models/dynamics/derived_state/docs/derived_state.pdf***

utils/relkin/docs/relkin.pdf

H.1 Introduction

The default vehicle states are available for the composite-body, core-body, and structural frames with respect to a specified inertial reference-frame. Additional states are available on request; the Derived States Model provides the following capabilities:

- Euler angles (angles only)
  - Euler angles of rotation from one frame to another frame
- LVLH (Frame determination)
  - Local-vertical, local-horizontal
  - Provides the LVLH frame
- NED (Frame determination)
  - North-East-Down
  - Provides the NED frame
- Orbital Elements representation
- Planet-fixed representation (Position only)
- Relative state representation (Translational and Rotational)
  - Used to provide the state of one frame relative to any other frame
- Solar Beta calculation (angle only)

H.1.1 Terminology

H.1.1.a Reference Frame

There are two distinct uses of reference-frame. The first is the one we have already seen in chapter D; it refers to a set of axes with an origin, oriented in some determinable sense. A vehicle (a *DynBody*) has 3 such reference-frames, see section B.5 for discussion of these. The second use of reference frame refers to one particular reference-frame, the reference reference-frame, the reference-frame relative to which the state is

referenced. In this document, we try to maintain the distinction by calling the latter the “reference reference-frame”, even though doing so is a little cumbersome.

H.1.1.b Subject

The subject is the object for which the state is generated. Frames have states; where we lazily refer to the “state” of a *DynBody*, it is usually safe to assume that the implication is that the value is the state of its *composite_body* frame.

H.1.2 S_define setup

Derived states are typically found in either the simulation-object that defines the subject reference-frame (typically the vehicle simulation-object), or, typical in simulations where there are many derived states, in a separate simulation-object devoted to derived states.

H.2 Euler Angles

Euler Angles are a set of three angles for expressing the orientation of one frame with respect to another. The three angles are represented in a 1x3 array representing rotations about the x, y, and z axes of a specified frame. Conventionally, these are called roll, pitch, and yaw (respectively), and often abbreviated to RPY (respectively). The order in which the axial-rotations are performed is important; JEOD provides the ability to use any of the 6 combinations of the three axes, plus another 6 combinations in which one axis is duplicated. However, not that if using the elements Yaw, Pitch, Roll, to specify the sequence then only the 6 non-duplicated cases are available. If using the axes, e.g. EulerXYZ, all twelve cases are available.

For example, the order of angles may be specified as *Roll_Pitch_Yaw*, *Yaw_Pitch_Roll*, *EulerXYZ*, *EulerXYX*, but not *Roll_Pitch_Roll*. Note – consecutive operations on the same axis are not an option in either case, since this is effectively only a single rotation through the combined angle.

H.2.1 S_define Setup

Specify an instance of the Euler Angles Derived State,

```
models/dynamics/derived_state/verif/SIM_Euler/S_define
 37 ##include "dynamics/derived_state/include/euler_derived_state.hh"
...
 46     jeod::EulerDerivedState euler_rpy;
```

and initialize it in one of two ways:

1. Assume that the reference reference-frame is the subject's parent in the reference-frame tree:

```
 62       P_DYN  ("initialization") euler_rpy.initialize ( dyn_body,
 63                                                         dyn_manager);
```

2. Specify the reference reference-frame explicitly:

```
64       P_DYN  ("initialization") euler_rpy_lvlh.initialize ( lvlh.lvlh_frame,
65                                                             dyn_body,
66                                                             dyn_manager);
```

H.2.2 Input File Setup

The three angles and the axes to which they are to be applied needs to be specified. The order can be any non-duplicating combination of Roll, Pitch, Yaw, with or without underscores (e.g. RollPitchYaw, Pitch_Yaw_Roll), or *Euler* followed by any non-repeating combination of X, Y, Z (e.g. *EulerXYZ*, *EulerYXY*)

models/dynamics/derived_state/verif/SIM_Euler/Modified_data/veh.py

```
 63   veh.euler_rpy.sequence = trick.Orientation.Roll_Pitch_Yaw
```

H.2.3 Loggable Data

The Euler angles are output as a 3-vector. Conventionally, the Euler sequence provided in the input file represents the rotations necessary to go from the reference reference-frame to the subject reference-frame. In this case, the angles are evaluated in the *ref_body_angles* variable (reference-to-body(subject)-angles). Under some circumstances, it is preferable, or additionally necessary, to obtain the Euler angles from the subject reference-frame to the reference reference-frame; these angles are provided with the *body_ref_angles* variable.

models/dynamics/derived_state/verif/SIM_Euler/Log_data/log_orbital_state_rec.py

```
 10   for ii in range(0,3) :
 11     dr_group.add_variable("veh.euler_pyr_lvlh.ref_body_angles[" + str(ii) + "]" )
 12     dr_group.add_variable("veh.euler_pyr_lvlh.body_ref_angles[" + str(ii) + "]" )
```

H.3 Orbital Elements

H.3.1 Introduction

The Orbital Elements derived state provides the state of a vehicle in terms of orbital elements. It provides the following data:

1. Orbital definition:
   (a) Semi-major axis (*semi_major_axis*)
   (b) Eccentricity (*e_mag*)
   (c) Semiparameter (semi-latus rectum) (*semiparam*)
   (d) Inclination (*inclination*)

(e) Argument of periapsis (*arg_periapsis*)

(f) Longitude of ascending node (*long_asc_node*)

2. Orbital characterizations:

(a) specific orbital energy (*orb_energy*)

(b) specific orbital angular momentum (*orb_ang_momentum*)

3. State definition:

(a) distance from focus (*r_mag*)

(b) relative speed (*vel_mag*)

(c) true anomaly (*true_anom*)

(d) mean anomaly (*mean_anom*)

(e) orbital anomaly (e.g. eccentric anomaly) (*orbital_anom*)

(f) mean motion (*mean_motion*)

WARNING: The calculation of orbital elements WILL be based on setting the identified planet (identified with *reference_name*) as the orbital focus. It will NOT consider proximity to that body, or any other celestial bodies, in verifying that the vehicle is, indeed, in orbit about that body. It WILL assume that the vehicle is in a Keplerian orbit about that body as the only other object in the universe.

Consequence – incorrect specification of *reference_name* will lead to nonsensical “orbital elements” data being generated relative to the specified planet when the vehicle is in orbit about another planet.

H.3.2 S_define Setup

The S_define setup includes three elements:

- Instance of the *OrbElemDerivedState*
- Initialization of the state, including specification of the subject reference-frame whose state is to be expressed in orbital elements
- Run-time update to generate the orbital elements of the subject reference-frame state.

The Orbital Elements Derived State is typically found either in the vehicle simulation-object, or in the relative-state or derived-state simulation-object (if one exists).

(based on models/dynamics/derived_state/verif/SIM_OrbElem/S_define)

```
 38 ##include "dynamics/derived_state/include/orb_elem_derived_state.hh"
...
 44     jeod::OrbElemDerivedState orb_elem;
...
 50       P_DYN  ("initialization") orb_elem.initialize ( dyn_body,
 51                                                        dyn_manager);
 52       (DYNAMICS, "environment") orb_elem.update ( );
```

H.3.3 Input File Setup

The only input-level specification is the identification of the central body that defines the focus of the Keplerian orbit for the purpose of deriving the orbital elements. This is identified by specifying *reference_name*.

models/dynamics/derived_state/verif/SIM_OrbElem/SET_test/RUN_ecc/input.py

```
 18 veh.orb_elem.reference_name = "Earth"
```

H.3.4 Loggable Data

All of the elements listed in the introduction to the Orbital Elements Derived State can be logged when prefixed with <*simulation-object*>.<*instance-of-OrbElemDerivedState*>.*elements*. For example:

models/dynamics/derived_state/verif/SIM_OrbElem/Log_data/log_orbital_state_rec.py

```
  8   dr_group.add_variable(   "veh.orb_elem.elements.semi_major_axis")
  9   dr_group.add_variable(   "veh.orb_elem.elements.semiparam")
```

Exercise 33. Revisiting an Orbital-Elements Initialized Simulation

H.4 Planet-fixed State

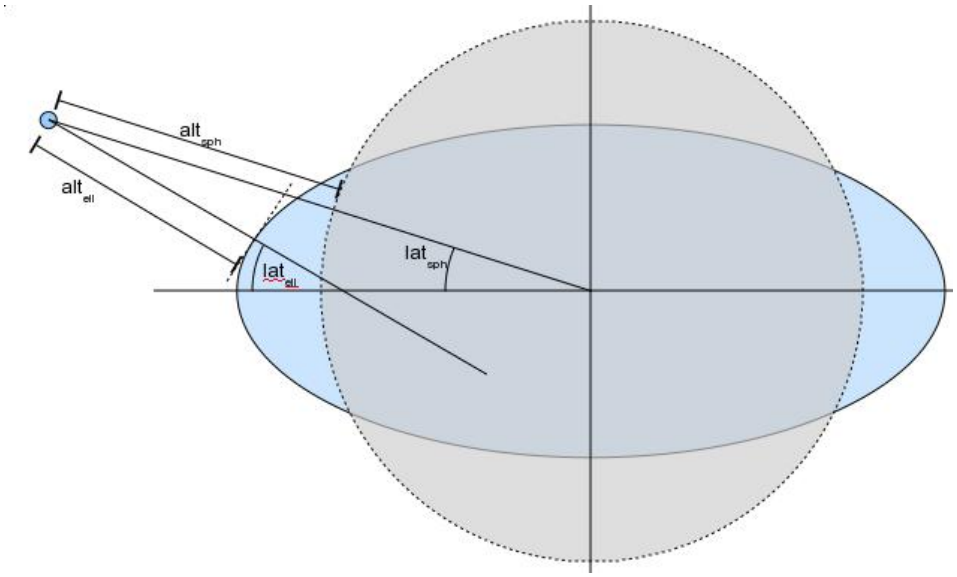
H.4.1 Introduction

The Planet-fixed Derived State provides translational position of some body's *composite_body* reference-frame, expressed relative to a planet-fixed reference-frame (rotating with the planet). Note that only the translational-position component of state is provided – no velocity, no rotational component. The position is available in three formats:

- Cartesian:
  - Three linear values centered on, and fixed to the planet (co-rotating)
- Spherical:
  - Two angular and one linear value that correspond to latitude, longitude, and altitude
- Elliptical:
  - Similar to spherical, but assumes an elliptical planet

Note – all three forms are provided, all of the time. Unlike the NED derived state, it is not necessary to declare the desired representation of the planetary state.

The difference between elliptical and spherical is illustrated in the model documentation, and reproduced here:



H.4.2 S_define Setup

The S_define setup includes three elements:

- Instance of the *PlanetaryDerivedState*
- Initialization of the state, with the specification of the subject body whose composite-body reference-frame's state will be evaluated.
- Run-time update to the state.

The Planet-fixed Derived State is typically found either in the vehicle simulation-object, or in the relative-state or derived-state simulation-object (if one exists).

(based on models/dynamics/derived_state/verif/SIM_Planetary/S_define)

```
 38 ##include "dynamics/derived_state/include/planetary_derived_state.hh"
...
 44     jeod::PlanetaryDerivedState pfix;
...
 50       P_DYN  ("initialization") pfix.initialize ( dyn_body,
 51                                                   dyn_manager);
...
 52       (DYNAMICS, "environment") pfix.update ( );
```

H.4.3 Input File Setup

The only input-level specification is the identification of the planet whose *pfix* reference-frame (all planets have a *pfix* reference-frame automatically) will serve as the basis for the calculation of the subject-body's state. This planet is identified by specifying *reference_name*.

models/dynamics/derived_state/verif/SIM_Planetary/SET_test/RUN_GEO/input.py

```
 94 veh.pfix.reference_name = "Earth"
```

H.4.4 Loggable Data

Relative to the planet-fixed frame, the vehicle state can be expressed in cartesian coordinates (3-vector), or elliptical or spherical coordinates (altitude, latitude, longitude).

models/dynamics/derived_state/verif/SIM_Planetary/Log_data/log_orbital_state_rec.py

```
 10   for ii in range(0,3) :
        dr_group.add_variable("veh.pfix.state.cart_coords[" + str(ii) + "]" )
 11   dr_group.add_variable(   "veh.pfix.state.sphere_coords.altitude")
 12   dr_group.add_variable(   "veh.pfix.state.sphere_coords.latitude")
 13   dr_group.add_variable(   "veh.pfix.state.sphere_coords.longitude")
 14   dr_group.add_variable(   "veh.pfix.state.ellip_coords.altitude")
 15   dr_group.add_variable(   "veh.pfix.state.ellip_coords.latitude")
 16   dr_group.add_variable(   "veh.pfix.state.ellip_coords.longitude")
```

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

H.5 Relative State

H.5.1 Introduction

The Relative State is perhaps the most utilized of the derived states. It provides the state of one reference frame relative to some other frame. It provides the full 12-state:

- *relative_state.trans.position*
- *relative_state.trans.velocity*
- *relative_state.rot.T_parent_this*
- *relative_state.rot.ang_vel_this*

The Relative State uses two identified reference frames, one called *subject* (as with other derived states), and another called *target*. The model is capable of obtaining the state of *target* relative to *subject* and that of *subject* relative to *target*, so the intended 'direction' of the relative state does not dictate which frame is which.

WARNING!

There is, nevertheless, a distinct difference between *target* and *subject*, and it is mostly hidden. The *target* reference-frame is of type *RefFrame*, while the *subject* reference-frame is of type *BodyRefFrame*. *BodyRefFrame* is a particular type of *RefFrame* (it inherits from *RefFrame*), so an instance of *BodyRefFrame* qualifies for use as either the *target* or *subject* reference-frame. However, an instance of *RefFrame* is **not** a *BodyRefFrame*, and **can only be used as the *target* reference-frame**.

Generally, any reference frame on a vehicle (including things like composite-body, structure, frames associated with attached *MassPoint* instances) is going to be a *BodyRefFrame*. Reference Frames used for reference

purposes, such as inertial reference frames, LVLH or NED frames, planet-fixed frames, etc. are NOT going to be *BodyRefFrame* instances. There are two consequences:

1. This derived state cannot be used to obtain the relative state of two frames not associated with any vehicles, such as that of another planet in an LVLH frame.

2. When setting up a relative state involving one *BodyRefFrame* and one generic *RefFrame* (or derivative of *RefFrame* that differs from *BodyRefFrame*), the generic *RefFrame* must **always** be assigned as the *target*.

H.5.1.a Relative Derived State and RefFrameState [optional]

An important limitation of the Relative State sub-model is that it can only provide the relative state of one frame relative to another. This is stored in the class *RefFrameState*, the methods of which apply a rigid interpretation to the values so created. The state of frame A relative to frame B has the following components:

- a position that is expressed in frame B
- a velocity that is expressed in frame B
- a transformation matrix from B to A (frame independent, representing the orientation of A relative to B)
- a quaternion set using the axes of frame B as reference to represent the orientation of A relative to B (equivalently, the transformation from B to A).
- an angular rate that is expressed in frame A.

Note – the quaternion set provided here is a left-handed transformation quaternion (as it is throughout JEOD). A left-handed transformation quaternion $\left(\vec{x}_A = Q_{B\to A}\vec{x}_B Q^*_{B\to A}\right)$ is equivalent to a right-handed rotation quaternion and conjugate to a left-handed rotation quaternion or a right-handed transformation quaternion. For integrating with external packages, it is important to know how the quaternion output will be interpreted.

Any attempt to interpret these values in any other way will lead to erroneous data. So, for example, the Relative State CANNOT express the position of frame A with respect to frame B in frame A, or in any third-frame (e.g. the position of a lidar unit relative to some reflector in a vehicle's LVLH reference frame is NOT possible directly from the Relative State sub-model).

Such data is, however, easily obtainable. First, the position of frame A relative to frame B can be expressed in frame B. Next, the orientation of frame B relative to frame C can be determined. These can both be obtained from (separate instantiations of) the Relative State sub-model. Next, applying the transformation matrix from frame B to frame C to the position vector gives the position vector expressed in frame C. Note, however, that this new position vector (A relative to B in C) MUST NEVER be assigned to the position variable of a *RefFrameState*, because it does not represent what *RefFrameState.position* is intended to represent. It must remain as a stand-alone data variable.

With a solid understanding of dynamics, then in some situations it is possible to circumvent this limitation by adding a *MassPointInit* (and consequentially a *BodyRefFrame*), M, to a vehicle such that it is co-located with frame B, and co-oriented with frame C. Then, finding the relative state of frame A with respect to frame M provides the relative position of frame A with respect to frame B, expressed using axes that are aligned with

frame C. This is only useful when frame C has a constant orientation relative to the vehicle structural frame (e.g. the vehicle structural frame, but not the LVLH frame), and care must be taken to ensure that the other resulting components of the *RefFrameState* are interpreted appropriately (e.g. angular rate will be of frame A with respect to frame C, expressed in frame A, and translational velocity will include the effects of the relative rotation of frame C relative to frame A over the moment arm from frame C to frame B).

H.5.2 S_define Setup

The *RelativeDerivedState* can be managed in two ways – either as a stand-alone implementation, updated by the simulation engine, or as one of a collection of *RelativeDerivedState* instances, managed by the Relative Kinematics model (see section H.10). Here, we illustrate the stand-alone implementation; the relative-kinematics managed implementation is illustrated in section H.10.

The S_define setup includes three elements in the stand-alone implementation:

- Instance of the *RelativeDerivedState*
- Initialization of the state, with the specification of the subject body. The subject frame (to be defined later) must exist as a frame in this body. The target frame (also to be specified later) is independent of this specification.
- Run-time update to the state.

(based on models/dynamics/derived_state/verif/SIM_Relative/S_define)

```
 36 ##include "dynamics/derived_state/include/relative_derived_state.hh"
...
 47     jeod::RelativeDerivedState  vehA_wrt_vehB_in_B;
...
 60     P_DYN  ("initialization") vehA_wrt_vehB_in_B.initialize ( vehA_body,
                                                                  dyn_manager);
...
 66       (DYNAMICS, "environment") vehA_wrt_vehB_in_B.update ( );
```

Note that if the *target* (or *subject*) reference-frame is generated by the Derived State Model (e.g. NED), the initialization and update methods for this relative state must be called after those for the other Derived State. Consequently (to ensure that the Relative State instances come last), where there is need for a Relative State there is often a simulation-object towards the end of the S_define dedicated to either derived states in general, or relative states specifically.

H.5.3 Input File Setup

The relative state is taken between two reference-frames, identified by their frame-names. Frame names are auto-constructed based on the name of the parent in the reference-frame tree. For information on frame-names, see section D.2.2.a.

Unlike some of the other derived states, there is no assumption made that the *subject* reference-frame is the composite-body frame of the subject-body, although it must be a frame associated with the subject-body (note – subject-body is identified with the state initialization method call in the S_define). Consequently, the *subject* reference-frame must be specified, along with the *target* reference-frame that is unique to the Relative State.

Both are identified by name, both must already have been named (equivalently, be registered with the Dynamics Manager)

```
models/dynamics/derived_state/verif/SIM_Relative/SET_test/RUN_AB_rot_AB_trans/input.py
 99 rel_state.vehicleA_wrt_vehicleB_in_B.subject_frame_name = "vehicleA.PointA"
100 rel_state.vehicleA_wrt_vehicleB_in_B.target_frame_name  = "vehicleB.PointB"
```

Additionally, the name of this state may be specified (optional)

```
 98 rel_state.vehicleA_wrt_vehicleB_in_B.name              = "A_wrt_B_in_B"
```

It may appear that requiring that the *subject* reference-frame must exist in the subject-body would restrict the relative state to only being able to provide the state of a *DynBody* object's frame from some other frame. Doing so would seriously limit the capabilities of the model, but that limitation is easily circumvented.

The value *direction_sense* specifies whether the relative state provided should be interpreted as either::

- the state of the subject reference-frame with respect to target reference-frame, expressed in the target reference-frame (*ComputeSubjectStateinTarget*), or
- the state of the target reference-frame with respect to subject reference-frame, expressed in the subject reference-frame (*ComputeTargetStateinSubject*)

Since there is no restriction imposed on the target reference-frame, this flexibility provides the capability of expressing the relative state of a non-*DynBody* state relative to a *DynBody* state.

Note that this value must be specified. The default setting is *undefined.*

```
101 rel_state.vehicleA_wrt_vehicleB_in_B.direction_sense =
                                    trick.RelativeDerivedState.ComputeSubjectStateinTarget
```

H.5.4 Relative State Between Points

Relative State can also find the state between any two named points on a vehicle. Recall, from section D.4.4, that *MassPoint* instances represent specific points on *MassBody* instances and *DynBody* instances. In section E.2.2, we covered the nature of the *add_mass_point* method, describing that when *MassPointInit* instances are added to a *DynBody*, they produce new instances of the *BodyRefFrame* class. This process is typically performed with the vehicle mass initialization process, as described in section E.1.2.

The result is that by adding these points at the initialization of the vehicle, new reference-frames are created; since this *RelativeState* Derived State can take the relative state between any two frames (subject to at least one of them being a *BodyRefFrame*), we can identify the relative state of any point on the vehicle relative to any other point on any other vehicle, or any other reference-frame anywhere in the simulation. The only limitation is that the relative state must be expressed in the reference-frame of one of the two points.

The names of the reference frames associated with these BodyRefFrames are constructed to be *<vehicle-name>.<point-name>*

H.5.5 Loggable Data

The standard *RefFrameState* values are associated with all *RelativeDerivedState* instances, as described in the introduction section to the *RelativeDerivedState.*

```
Example
for ii in range(0,3) :
 dr_group.add_variable("rel_state.A_wrt_B.rel_state.trans.position[" + str(ii) + "]" )
 dr_group.add_variable("rel_state.A_wrt_B.rel_state.trans.velocity[" + str(ii) + "]" )
 dr_group.add_variable("rel_state.A_wrt_B.rel_state.rot.ang_vel_this[" + str(ii) + "]" )
 for jj in range (0,3) :
   dr_group.add_variable("rel_state.A_wrt_B.rel_state.rot.T_parent_this
                                                    [" + str(ii) + "][" + str(ii) + "]" )
```

Exercise 35. Derived States Exercise - Relative State

H.6 Local-Vertical-Local-Horizontal (LVLH)

H.6.1 Introduction

The LVLH derived state comes in two components – the definition of the LVLH frame (*LvlhFrame*), and the state relative to that frame (*LvlhRelativeDerivedState*).

For example, consider a vehicle docking with ISS. It may be desirable to know the state of the vehicle relative to the ISS LVLH reference-frame. The *LvlhFrame* instance provides the ISS LVLH frame. The *LvlhRelativeDerivedState* instance provides the state relative to that frame.

H.6.1.a LvlhFrame

Note that the *LvlhFrame* (as with all reference-frames) does have its own state (lvlh_frame.state, a conventional *RefFrameState*). Do not confuse the state **of** the LVLH reference-frame for the state of a vehicle expressed **in** the LVLH reference-frame.

The LvlhFrame model provides a **rectilinear** interpretation of LVLH, with axes defined as follows:

z: negative radial

y: negative orbital angular momentum vector

x: completes right-hand system (projection of velocity onto horizontal)

Notes:

3. The frame represents an instantaneous ‘snapshot’; its position and orientation typically change constantly. The frame is rectilinear (as are all JEOD frames), but the state relative to that frame may be expressed in curvilinear.

4. A common misconception is to equate the x-axis with the direction of the velocity vector. In cases of near-circular orbits, the velocity vector is close to the x-axis, but that is not its definition.

The instance of *LvlhFrame* must be attached to another reference frame. It may be associated (co-located) with any other JEOD reference frame. That could be, for example, the frame associated with a MassPoint such as a (i.e. has the same origin as) the composite-body frame of a specified subject_body (*DynBody*). It further uses a planet, identified by name as *reference_name* to define vertical and horizontal.

Exercise 36. Derived States Exercise – Define a LVLH Reference Frame

H.6.1.b LvlhRelativeDerivedState

The *LvlhRelativeDerivedState* inherits directly from the *RelativeDerivedState* class (see section H.5). As such, it has associations to two reference frames, known as the *subject* and the *target*.

When implementing the *LvlhRelativeDerivedState* to represent a state of a vehicle relative to some LVLH frame, it is important to make the *LvlhFrame* the *target* frame (as opposed to the *subject* frame). While it is still possible (in principle) to set the direction to be of-target-with-respect-to-subject, doing so would be largely pointless; the intent is to provide the state of some frame relative to a specified *LvlhFrame*.

The *subject* frame in a *RelativeDerivedState* must be associated with the subject-body, which is assigned at initialization. The *target* frame in a *RelativeDerivedState* can be any frame in the simulation (although for this case, it really should be a *LvlhFrame* – although no explicit type-verification is performed).

An astute reader may, at this point question the difference between the RelativeDerivedState and the LvlhRelativeDerivedState – which gets used, and when? It is certainly possible to use a *RelativeDerivedState* to get the state of a frame relative to the *LvlhFrame*. However, the *LvlhRelativeDerivedState* adds the capability to switch output format between rectilinear (which is all you get with *RelativeDerivedState*) and circular-curvilinear representations. Because this additional feature comes with minimal overhead, it is the recommended practice going forward for all states that are formed relative to a *LvlhFrame*.

H.6.2 S_define Setup

Because it provides a reference-frame (rather than a state), the instance of *LvlhFrame* is often found in the vehicle simulation-object rather than in the relative-state simulation-object. The instance of *LvlhRelativeDerivedState* may be found in either the relative-state simulation-object (where it exists) or the vehicle simulation-object.

H.6.2.a LvlhFrame

The S_define setup starts with adding an instance of *LvlhFrame* to the vehicle with which the frame will be associated. Initialization requires knowledge of the Dynamics Manager. Run-time execution requires no further arguments.

```
models/dynamics/derived_state/verif/SIM_LvlhRelative/S_define
    88  ##include "utils/lvlh_frame/include/lvlh_frame.hh"
...
    92    jeod::LvlhFrame               lvlh_frame;
...
    98        P_ENV ("initialization") lvlh_frame.initialize (dyn_manager);
    99              ("initialization") lvlh_frame.update ();
...
   102        (DYNAMICS, "environment") lvlh_frame.update ();
```

H.6.2.b LvlhRelativeDerivedState

Instantiation:

```
models/dynamics/derived_state/verif/SIM_LvlhRelative/S_define
   122 ##include "dynamics/derived_state/include/lvlh_relative_derived_state.hh"
...
   136    jeod::LvlhRelativeDerivedState  vehB_in_vehA_rectilvlh;
```

Initialization:

The initialize method takes a reference to the subject-body (that is the one that is *not* attached to the *LvlhFrame*) and to the Dynamics Manager.

```
   159        P_DYN ("initialization") vehB_in_vehA_rectilvlh.initialize ( vehB_body,
   160                                                                     dyn_manager);
```

Execution:

Simple update call:

```
  174         (DYNAMICS, "environment") vehB_in_vehA_rectilvlh.update ();
```

H.6.3 Input File Setup

H.6.3.a LvlhFrame

It is necessary to specify both the reference frame to which the *LvlhFrame* will be anchored, and the planet that will be used to define local-vertical.

```
models/dynamics/derived_state/verif/SIM_LvlhRelative/SET_test/RUN_test0/input.py
    40 vehA.lvlh_frame.subject_name    = "vehicleA.composite_body"
    41 vehA.lvlh_frame.planet_name     = "ref_planet"
```

At initialization, the LvlhFrame will be fully constructed and named based on these settings

<subject-name>.<planet-name>.lvlh

In this example, that becomes:

vehicleA.composite_body.ref_planet.lvlh

H.6.3.b LvlhRelativeDerivedState

The LvlhRelativeDerivedState is configured with the following settings:

- **name**. A user-specified arbitrary name
- **subject-frame-name**. This is the name of the frame whose state is to be compared against the LVLH frame's state.
- **target_frame_name**. This is the name of the frame that will be constructed by the intended LVLHFrame instance, i.e. something like <vehicle-name>.<body-frame-name>.<planet-name>.lvlh
- lvlh_type. This is where the user can choose between:
  - Rectinlinear (default)
  - CircularCurvilinear
  - EllipticalCurvilinear (present, but not supported. **Do not use this option**)

```
 58 rel_state.vehB_in_vehA_rectilvlh.name              = "B_in_A_rectilvlh"
 59 rel_state.vehB_in_vehA_rectilvlh.subject_frame_name = "vehicleB.composite_body"
 60 rel_state.vehB_in_vehA_rectilvlh.target_frame_name  = "vehicleA.composite_body.ref_planet.lvlh"
 61 rel_state.vehB_in_vehA_rectilvlh.lvlh_type          = trick.LvlhType.Rectilinear
```

NOTES:

1. **Initialization: at the time that this LVLH-relative-state gets initialized, the target-frame-name must have already been constructed, and that is done at the initialization of the respective *LvlhFrame* instance. Therefore, it is recommended that either or both of the the following steps be taken:**
   a. **The LVLH-relative-state is put in a simulation-object that appears later in the S_define than the simulation-object that contains the LvlhFrame instance that it targets.**
   b. **The initialization of the LVLH-relative-state is given a lower priority (happens later) than that of the LvlhFrame instance that it targets.**
2. **Updates: as with regular Relative-states, at the time that this LVLH-relative-state gets updated, both the subject-frame and target-frame should be in a time-consistent state. The subject-frame is typically a body-based frame (typically updated with the state integration), but now the target-frame is typically a *LvlhFrame* instance, which is typically updated as an environment-class job. Care should**

be taken to ensure that the update of the LVLH-relative-state be scheduled to occur AFTER the scheduled update to the target-frame.

H.6.4 Loggable Data

The *LvlhFrame* state is typically a utility component. While the state of the LVLH reference-frame can be logged, it is more typical to log only the output of the *LvlhRelativeDerivedState*. As with RelativeDerivedState (see section H.5), the *LvlhRelativeDerivedState* has a *RefFrameState* called *rel_state.* This has the standard state descriptors:

```
for ii in range(0,3) :
  dr_group.add_variable("rel_state.vehA_in_vehB_rectilvlh.rel_state.trans.position["+str(ii)+"]")
  dr_group.add_variable("rel_state.vehA_in_vehB_rectilvlh.rel_state.trans.velocity["+str(ii)+"]")
  dr_group.add_variable("rel_state.vehA_in_vehB_rectilvlh.rel_state.rot.ang_vel_this["+str(ii)+"]")

for ii in range(0,3) :
 for jj in range (0,3) :
   dr_group.add_variable("rel_state.vehA_in_vehB_rectilvlh.rel_state.rot.T_parent_this["+str(ii)+"]["+str(jj)+"]")
```

Exercise 37. Derived States Exercise – Relative LVLH State

H.7 Local-Vertical-Local-Horizontal (LVLH)

H.7.1 Introduction

A very important realization is that this Derived State instance does not provide a state, it provides a new reference-frame, from which a state can be derived.

For example, consider a vehicle docking with ISS. It may be desirable to know the state of the vehicle relative to the ISS LVLH reference-frame. This derived state provides the ISS LVLH frame. The Relative Derived State (see section H.5) can be used to obtain the state of one of the vehicle's reference-frames relative to this new LVLH frame.

Note that this new reference-frame (as with all reference-frames) does have its own state (lvlh_frame.state, a conventional *RefFrameState*). Do not confuse the state **of** the LVLH reference-frame for the state of a vehicle expressed **in** the LVLH reference-frame.

The default setting for the LVLH frame is to provide a **rectilinear** interpretation, with axes defined as follows:

z: negative radial

y: negative orbital angular momentum vector

x: completes right-hand system (projection of velocity onto horizontal)

Notes:

1. This is rectilinear only.

2. A common misconception is to equate the x-axis with the direction of the velocity vector. In cases of near-circular orbits, the velocity vector is close to the x-axis, but that is not its definition).

The LVLH frame is co-located with (i.e. has the same origin as) the composite-body frame of a specified subject_body (*DynBody*). It further uses a planet, identified by name as *reference_name* to define vertical and horizontal.

When implementing the *RelativeDerivedState* (see section H.5) to represent a state of a vehicle relative to the LVLH frame, it is important to make the LVLH frame the *target_frame* (as opposed to the *subject_frame*) regardless of the 'direction' of the relative state.

H.7.2 S_define Setup

Because it provides a reference-frame (rather than a state), the instance of LvlhDerivedState is often found in the vehicle simulation-object rather than in the relative-state simulation-object.

The S_define setup starts with an instance of the *LVLHDerivedState*. It is initialized by specifying the subject body (identifying the reference frame with common origin with the new LVLH frame). At run-time, it is updated as needed. In the example below, the frame's state is updated at the DYNAMICS rate as an environment-class job. This step is only necessary if the LVLH frame's state is changing – i.e. if the origin of the anchoring subject reference-frame is moving in inertial space – but this is usually the case.

(based on models/dynamics/derived_state/verif/SIM_LVLH/S_define)

```
 41 ##include "dynamics/derived_state/include/lvlh_derived_state.hh"
...
 48     jeod::LvlhDerivedState lvlh;
...
 55       P_DYN  ("initialization") lvlh.initialize ( dyn_body,
 56                                                   dyn_manager);
 57       (DYNAMICS, "environment") lvlh.update ( );
```

H.7.3 Input File Setup

It is necessary only to specify the planet reference:

models/dynamics/derived_state/verif/SIM_LVLH/Modified_data/veh.py

```
 20   veh.lvlh.planet_name = "Earth"
```

Note – the LVLH reference-frame is named <*subject*>.<*reference_name*>.*lvlh*, for example, *station.Earth.lvlh* would be the LVLH reference frame associated with a vehicle called *station*.

H.7.4 Loggable Data

The LVLH state is typically a utility component. While the state of the LVLH reference-frame can be logged, it is more typical to log the state of a vehicle relative to this new reference-frame. This is covered in the Relative State section (see section H.5).

H.8 North-East-Down (NED)

Additional Document Reference: *utils/planet_fixed/docs/planet_fixed.pdf*

H.8.1 Introduction

Similar to the LVLH derived state (previous section), the North-East-Down component of the Derived State Model does not provide a state relative to a NED frame. It provides a reference-frame, from which a state can be derived. It is typically used in conjunction with *RelativeDerivedState* to derive a state relative to this NED reference-frame. The NED frame may be attached to a vehicle, or to some fixed structure, such as a launch-site.

As with all reference-frames, the new NED reference-frame has its own state, expressed:

- with respect to the planet-fixed system (*ned_state.sphere_coords*, *ned_state.ellip_coords*, *ned_state.cart_coords*; these are components of the *NorthEastDown* class which is a derivative of the *PlanetFixedPosition* class, see section H.4 for details), and
- with respect to inertial space *(ned_state.ned_frame.state;* this is a *RefFrameState*, see section D.2 for details).

Again, neither represents the state of a vehicle in the NED frame.

The NED frame (*ned_state.ned_frame*) configuration follows these rules:

- It is centered on the *composite_body* frame of a specified *subject_body* (*DynBody*)
- It uses a planet, identified by name as *reference_name* to define North, East, and Down.
- The axes are rectilinear, and are aligned with either the local spherical coordinates or local elliptical coordinate-axes of a planet (see section H.4 for this distinction). The choice is set by the user with the variable *altlatlong_type*.
- The *NorthEastDown* class (*ned_state*) derives directly from *PlanetFixedPosition*; there is no need to implement an instance of *PlanetFixedPosition* to support the NED frame.
- When implementing the *RelativeDerivedState* (see section H.5) to represent a state relative to the NED frame, it is important to make the NED frame the *target_frame* (as opposed to the *subject_frame*) regardless of the 'direction' of the relative state.

H.8.2 S_define Setup

Because it provides a reference-frame (rather than a state), the instance of NedDerivedState is often found in the vehicle or planet simulation-object rather than in the relative-state simulation-object. The S_define setup follows the same outline as that for the LVLH frame:

- Instantiation of the frame
- Initialization of the frame, with specification of the subject-body.

- Optional run-time update of the frame's state, necessary only if the frame itself is moving through planet-fixed space.

```
(based on models/dynamics/derived_state/verif/SIM_NED/S_define)
 41 ##include "dynamics/derived_state/include/ned_derived_state.hh"
...
 47     NedDerivedState ned;
...
 54       P_DYN  ("initialization") ned.initialize ( dyn_body,
 55                                                  dyn_manager);
...
 56       (DYNAMICS, "environment") ned.update ( );
```

H.8.3 Input File Setup

It is necessary to declare:

- the planet reference

```
models/dynamics/derived_state/verif/SIM_NED/Modified_data/veh.py
 68   veh.ned.reference_name = "Earth"
```

- the intended interpretation of the frame state; the NED implementation can be aligned with either the local elliptical or a spherical coordinate-axes The difference between elliptical and spherical is illustrated in the model documentation, and reproduced in the Planet-fixed Derived States section (section H.4).

```
models/dynamics/derived_state/verif/SIM_NED/SET_test/RUN_ell_equ/input.py
 49 veh.ned.ned_state.altlatlong_type = trick.NorthEastDown.elliptical
```

```
models/dynamics/derived_state/verif/SIM_NED/SET_test/RUN_sph_equ/input.py
50 vehA.ned.ned_state.altlatlong_type = trick.NorthEastDown.spherical
```

The frame created by this implementation is named <subject>.<reference_name>.ned. For example, *station.Earth.ned* would be the NED reference-frame attached to a vehicle called *station* referenced to *Earth.*

H.8.4 Loggable Data

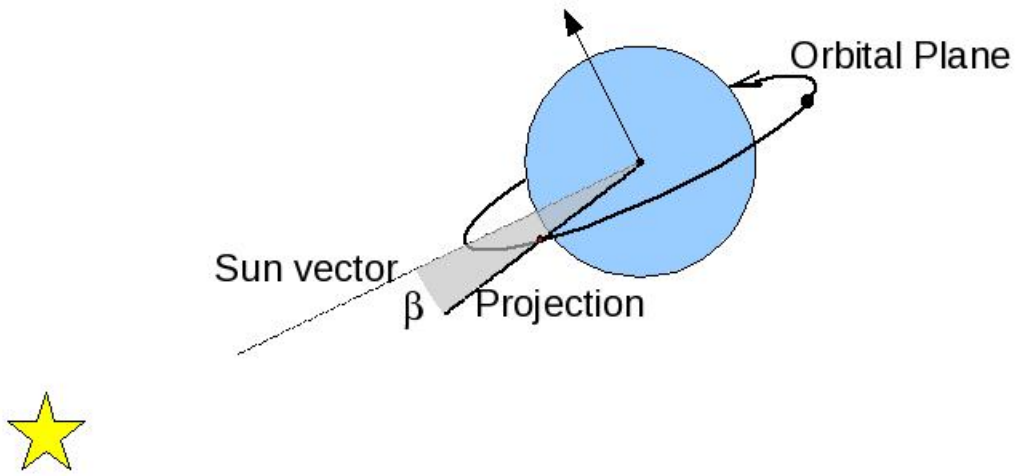
There is typically nothing to log from the North-East-Down derived state. Logging of vehicular states relative to this reference-frame is described in the Relative Derived State (section H.5). The state of the frame itself may be expressed in either cartesian, elliptical, or spherical coordinates relative to the planet-fixed reference frame, or in inertial coordinates, relative to the inertial reference frame. However, these same data can be acquired by implementing a Planet-fixed Derived State for the vehicle's composite-body reference-frame.

Example

```
for ii in range(0,3) :
 dr_group.add_variable("veh.ned.ned_state.cart_coords[" + str(ii) + "]" )
 dr_group.add_variable("veh.ned.ned_state.ned_frame.state.trans.position["+str(ii)+"]" )
 dr_group.add_variable("veh.ned.ned_state.ned_frame.state.trans.velocity["+str(ii)+"]" )
dr_group.add_variable("veh.ned.ned_state.sphere_coords.altitude")
dr_group.add_variable("veh.ned.ned_state.sphere_coords.latitude")
dr_group.add_variable("veh.ned.ned_state.sphere_coords.longitude")
dr_group.add_variable("veh.ned.ned_state.ellip_coords.altitude")
dr_group.add_variable("veh.ned.ned_state.ellip_coords.latitude")
dr_group.add_variable("veh.ned.ned_state.ellip_coords.longitude")
```

H.9 Solar Beta

The Solar Beta derived state provides the angle between the orbital plane and the planet-to-sun vector or, in other words, the angle between the planet-to-sun vector and the projection of the planet-to-sun vector onto the orbital plane.



This derived state provides only the solar-beta angle.

H.9.1 S_define Setup

The S_define setup includes three elements:

- Instance of the *SolarBetaDerivedState*
- Initialization of the state, with the specification of the subject body (vehicle) that will be used to define the orbital plane.
- Run-time update to the state.

(based on models/dynamics/derived_state/verif/SIM_SolarBeta/S_define)

```
 42 ##include "dynamics/derived_state/include/solar_beta_derived_state.hh"
...
 48     SolarBetaDerivedState solar_beta;
...
 56       P_DYN  ("initialization") solar_beta.initialize ( dyn_body,
 57                                                         dyn_manager);
...
 58       (DYNAMICS, "environment") solar_beta.update ( );
```

H.9.2 Input File Setup

The only input-level specification is the identification of the planet which, with the subject-body specified in the S_define, will define the orbital plane. This planet is identified by specifying *reference_name*. It must be already registered with the Dynamics Manager.

models/dynamics/derived_state/verif/SIM_SolarBeta/SET_test/RUN_incl_0/input.py

```
 98 veh.solar_beta.reference_name = "Earth"
```

Note that the plane used in the calculation of the solar-beta angle is defined by the subject-body (specified in the S_define), the subject-body's inertial state, and the specified planet. If the subject body is not in the vicinity of the specified planet, and in orbit elsewhere in the Solar System, then the plane will still be defined by the same specified data points, even when doing so is essentially meaningless.

H.9.3 Loggable Data

The solar-beta angle is the only output:

models/dynamics/derived_state/verif/SIM_SolarBeta/Log_data/log_orbital_state_rec.py

```
 4   dr_group.add_variable(   "veh.solar_beta.solar_beta")
```

Exercise 38. Solar Beta

H.10 Management of States (Relative Kinematics Model)

Additional Document Reference: *utils/relkin/docs/relkin.pdf*

The Relative Kinematics (relkin for short) Model provides a management tool for managing multiple instances of *RelativeDerivedState* between:

- multiple points of interest
- on multiple vehicles
- using multiple reference frames

e.g. between lidar units and reflectors during docking procedures.

The Relative Kinematics Model is a management tool only. It does not provide any relative states by itself.

H.10.1 S_define Setup

Where the Relative Kinematics model is used, conventionally it is given its own simulation-object, along with all of the associated *RelativeDerivedState* instances that it manages, and possibly other *DerivedState* instances that it does not manage. This simulation-object is typically found towards the end of the JEOD-driven simulation-objects of an S_define.

Declare an instance of *RelativeKinematics* and all desired instances of *RelativeDerivedState*

```
models/dynamics/rel_kin/verif/SIM_RELKIN_VERIF/S_define

 48 ##include "dynamics/rel_kin/include/relative_kinematics.hh"
 49 ##include "dynamics/derived_state/include/relative_derived_state.hh"
...
 62     jeod::RelativeKinematics         relkin;
 63     jeod::RelativeDerivedState       cm_relstate;
 64     jeod::RelativeDerivedState       sensor1_relstate;
 65     jeod::RelativeDerivedState       sensor2_relstate;
```

Each *RelativeDerivedState* is initialized, and added to the Relative Kinematics Model.

```
 78       P_DYN  ("initialization") cm_relstate.initialize ( body2,
 79                                                           dyn_mgr);
 80       P_DYN  ("initialization") relkin.add_relstate ( cm_relstate);
...
etc.
```

Updating the states can be performed one of two ways:

- All registered states are updated together

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

- An individual state in the collection is updated. This process is based on the name of the state, and while each call is state-specific, multiple calls may be made to update multiple states. Typically, where it is necessary to make many individual calls, it is often more useful to allocate those states to a separate *RelativeKinematics* instance.

```
94       P_ENV ("derivative") relkin.update_single (cm_relstate.name.c_str());
```

Note - when using *update_single* AND *update_all* together, the affected states will be updated on both schedules.

In the example above, all states are updated at the DYNAMICS rate, and the *cm_relstate* instance is additionally updated at the derivative rate.

H.10.2 Input File Setup

As with the Relative Derived State (section H.5), the relative states must name the respective subject and target reference-frames, and specify the direction_sense. In addition, the formerly-optional naming of the state is now required.

models/dynamics/derived_state/verif/SIM_Relative/SET_test/RUN_AB_rot_AB_trans/input.py

```
 98 rel_state.vehicleA_wrt_vehicleB_in_B.name              = "A_wrt_B_in_B"
```

H.10.3 Loggable Data

The Relative Kinematics Model provides no additional data. See the Relative Derived State (section H.5) for instructions on logging the relative states.

Exercise 39. Managing Relative States with the Relkin Manager

Exercise 40. Managing Relative States with Relkin, part II

Chapter I Miscellaneous Topics

I.1 Earth Lighting

Document reference: *environment/earth_lighting/docs/earth_lighting.pdf*

I.1.1 Introduction

The Earth Lighting Model calculates the amount of light impinging a spacecraft, assuming that it is in low-Earth orbit. It is intended for production of simulation graphics, it is not intended to be used to simulate a physical force from radiation pressure.

It provides the lighting resulting from three sources -

- Direct Sunlight
- Sunlight reflected off Earth
- Sunlight reflected off Moon

using the vehicle position and the planetary positions (which are obtained from the Ephemeris Model).

I.1.2 Setting up the S_define

The Earth-Lighting Model is typically found in its own small simulation-object comprising the instantiation, one initialization, and one run-time call.

```
Example (pseudo-code)
##include "environment/earth_lighting/include/earth_lighting.hh"
...
class LightingSimObject: public Trick::SimObject {
…
    jeod::EarthLighting lighting;
LightingSimObject( … )
{
    P_ENV ("initialization") lighting.initialize(dyn_manager);

    (DYNAMICS, "scheduled") lighting.calc_lighting(<position>);
};
...
};
LightingSimObject lighting(...)
```

Before the *initialize* method is called, Earth, Moon, and Sun must already be registered with the Dynamics Manager.

The position value to be passed into the *calc_lighting* method is the vehicle (composite-body) position relative to the earth-centered inertial (ECI) reference-frame. For most cases, where the vehicle is integrated in this frame, that position would be *vehicle.dyn_body.composite_body.state.trans.position*:

```
Example (pseudo-code)
##include "environment/earth_lighting/include/earth_lighting.hh"
##include "utils/ref_frames/include/ref_frame_state.hh"
...
class LightingSimObject: public Trick::SimObject {
…
    jeod::EarthLighting lighting;
LightingSimObject( RefFrameTrans & veh_state_in)
:
   vehicle_body_trans_state(veh_state_in)
{
    P_ENV ("initialization") lighting.initialize(dyn_manager);

    (DYNAMICS, "scheduled") lighting.calc_lighting(vehicle_body_trans_state.position);
};
...
  RefFrameTrans &  vehicle_body_trans_state;
};
LightingSimObject lighting(vehicle.dyn_body.composite_body.state.trans)
```

In cases where the vehicle is not integrated in the ECI frame, and that position is not known for any other reason, then another call may be necessary to compute the position relative to the ECI frame. In this case, take the following steps:

- Pass in at construction:
  - a *RefFrame* (called *earth_frame*, for now) representing the ECI frame
  - A *BodyRefFrame* (called *vehicle_frame*, for now) representing the vehicle body-frame
- Instantiate a 3-vector in the simulation-object to accommodate the vehicle position in the ECI frame.
- Add a *compute_position_from* call right before the *calc_lighting* call

```
Example
 (DYNAMICS, "scheduled") vehicle_frame.compute_position_from( earth_frame, pos_rel_earth);
 (DYNAMICS, "scheduled") lighting.calc_lighting(pos_rel_earth);
```

I.1.3 Setting up the Input File

There are two parameters that must be set for the Earth Lighting Model to function correctly. Both the *sun_earth* and *moon_earth* structures have a value called *phase*, which represents the phase of the sun and moon respectively as viewed from Earth. Both default to 0.0, indicating that neither will illuminate the vehicle. To include solar and lunar illumination, set these values.

```
Example
lighting.lighting.moon_earth.phase = 0.5
lighting.lighting.sun_earth.phase = 1.0
```

Note that these values are associated with how much of the body is visible from Earth in general, NOT from the vehicle specifically. The *sun_earth.phase* value should always be 1.0. The *moon_earth.phase* value should be between 0.0 (new moon) and 1.0 (full-moon). The value 0.5 above represents a quarter-moon (that is, the quarter-phase, corresponding to half of the near-side of Moon being illuminated).

I.1.4 Logging Data

The output of the model is the lighting variable, representing the fraction of maximum illumination from each source.

```
(based on
models/environment/earth_lighting/verif/SIM_LIGHT_CIR/Log_data/LIGHT_verif_rec.py)
 16    dr_group.add_variable("LIGHT.lighting.sun_earth.lighting")
...
 20    dr_group.add_variable("LIGHT.lighting.moon_earth.lighting")
...
 24    dr_group.add_variable("LIGHT.lighting.earth_albedo.lighting”)
```

I.2 Orientation

Document reference: ***utils/orientation/docs/orientation.pdf***

I.2.1 Introduction

The Orientation Model is used to provide different techniques for representing the orientation of one reference-frame relative to another. For example, the body-action *MassBodyInit* uses this model to enable a user to specify the orientation of a *DynBody* object's body reference-frame with respect to the *DynBody* object's structural reference-frame.

To specify an orientation, the user must set the Orientation object's *data_source* data member to identify which of the four supported representations is being used, and must populate the data member(s) appropriate to that selected representation. The four options, and their respective data members are as presented below. With each

case is an example representing a 180-degree rotation about the x-axis; in these examples, *orientation* represents the instance of Orientation being defined.

1. Transformation matrix:

    1. Use option *Orientation*::*InputMatrix*

    2. Variable *trans* is a 3x3 array of doubles representing a transformation matrix with a determinant value of 1.

```
Example
orientation.data_source = Orientation::InputMatrix
orientation.trans[0] = [1, 0, 0]
orientation.trans[1] = [0,-1, 0]
orientation.trans[2] = [0, 0,-1]
```

    Note – orientation.trans may also be assigned thus

```
orientation.trans = [[1, 0, 0], [0,-1,0], [0,0,-1]]
```

    but it tends to be easier to read when entered one row at a time.

2. (Left transformation) quaternion:

    1. Use option *Orientation*::*InputQuaternion*

    2. Variable *quat* is of type *Quaternion* (see section I.6); it is interpreted as a unit- left- quaternion.

```
Example
orientation.data_source = Orientation::InputQuaternion
orientation.quat.scalar = 0
orientation.quat.vector = [1, 0, 0]
```

3. Eigen rotation:

    1. Use option *Orientation*::*InputEigenRotation*

    2. Variable *eigen_axis* provides the axis about which the rotation is to be made. It is a 3-array of doubles and should be a unit vector.
    Variable *eigen_angle* provides the angle of rotation about axis *eigen_axis*.

```
Example
orientation.data_source = Orientation::InputEigenRotation
orientation.eigen_axis = [1,0,0]
orientation.eigen_angle = trick.attach_units( "d", 180.0)
```

4. Euler rotation sequence:

    1. Use option *Orientation*::*InputEulerRotation*

    2. Variable *euler_sequence* is an enumerated value of type *EulerSequence*. It provides the sequence of axes about which rotations will be applied.

Variable *euler_angles* provides the angles for each of the three specified axes. It is a 3-array of doubles.

```
Example
orientation.data_source = Orientation::InputEulerRotation
orientation.euler_sequence  = trick.Orientation.Roll_Yaw_Pitch
orientation.euler_angles = trick.attach_units( "d", [180.0, 0.0, 0.0])
```

Note – the Orientation Model does not specify which reference-frame is oriented, nor to which reference-frame the values are referenced. The interpretation of these data is left to the model that invokes the Orientation Model. For example, when adding a *MassPoint* with the *MassBodyInit* body-action, each instance of *MassPointInit* has an instance of *Orientation* called *pt_orientation,* and an instance of *FrameSpec* indicating whether *pt_orientation* provides structure-to-point, or point-to-structure orientation; these are the only options for that process.

```
Example
vehicle_mass_init.points[0].pt_orientation.data_source = trick.Orientation.InputEigenRotation
vehicle_mass_init.points[0].pt_orientation.eigen_axis = [1,0,0]
vehicle_mass_init.points[0].pt_orientation.eigen_angle = trick.attach_units( "d", 180.0)
vehicle_mass_init.points[0].pt_frame_spec = trick.MassBasicPointInit.PointToStruct
```

I.2.1.a Addendum on Euler Angles

Euler Angles are a set of three angles for expressing the orientation of one frame with respect to another. The three angles are represented in a 3-array representing rotations about the x, y, and/or z axes of a specified frame. Conventionally, these are called roll, pitch, and yaw (respectively), and often abbreviated to RPY (respectively). The order in which the axial-rotations are performed is important; JEOD provides the ability to use any of the 6 combinations of the three axes, plus another 6 combinations in which one axis is duplicated. However, not that if using the elements Yaw, Pitch, Roll, to specify the sequence then only the 6 non-duplicated cases are available. If using the axes, e.g. EulerXYZ, all twelve cases are available.

For example, the order of angles, *euler_sequence* may be specified as:

```
Example
orientation.euler_sequence = trick.Orientation.Roll_Pitch_Yaw
orientation.euler_sequence = trick.Orientation.YawPitchRoll
orientation.euler_sequence = trick.Orientation.EulerXYZ
orientation.euler_sequence = trick.Orientation.EulerXYX
```

Conversely, the following example is NOT allowed:

```
~~orientation.euler_sequence = trick.Orientation.Roll_Pitch_Roll.~~
```

Notes:

1. Consecutive operations on the same axis are not an option in either case, since this is effectively only a single rotation through the combined angle.

2. Yaw, Pitch Roll, may be separated with an underscore or pushed together, as shown in the first two examples above.

I.3 Propagated Planet

Document reference: *environment/ephemerides/docs/ephemerides.pdf*

I.3.1 Introduction

The Propagated Planet sub-model of the Ephemerides Model provides the ability to simulate a dynamic body for which ephemeris data is not available, yet which has mass sufficient to warrant a gravity model. It is particularly suited to simulations involving asteroids and other minor planets.

The model application is not limited to those bodies for which ephemeris data is not available. The ephemeris data used to populate a planetary state is empirically collected, while the simulation data used to populate a vehicle state is numerically integrated. There are scenarios where it is desirable to ensure that the planet and vehicle are subject to the same environmental factors - the same gravitational perturbations, etc. - and for which it is therefore desirable to integrate the planetary state rather than use observed data.

I.3.2 S_define

The simulation-object for a propagated-planet is, as one might expect, a blend of those found for a regular planet and for a vehicle, with some additional components.

I.3.2.a Planet-based Components

From the regular planet simulation-object (e.g. *earth_basic.sm*), it is still necessary to instantiate examples of the following objects:

- Planet
- Gravity Source

models/environment/ephemerides/verif/SIM_prop_planet/S_define

```
201 ##include "environment/planet/include/planet.hh"
...
203 ##include "environment/gravity/include/spherical_harmonics_gravity_source.hh"
...
217    jeod::Planet planet;
...
219    jeod::SphericalHarmonicsGravitySource gravity_source;
```

Then make the associated initialization and update calls.

```
253       P_ENV  ("initialization") gravity_source.initialize_body ( );
254
255       P_ENV  ("initialization") gravity_manager.add_grav_ source (gravity_source);
256
257       P_ENV  ("initialization") planet.register_model (
258           gravity_sourcey, dyn_manager);
259
260       P_BODY  ("initialization") planet.initialize ( );
```

Note that gravity*_manager* is passed in at construction, just as in *earth_basic.sm.*

The propagated planet must be populated with suitable data; for a major planet this is typically performed with default data (since these models are well known). For a minor planet, this *may* be performed with default data, or for a simulation-specific body, the necessary values may be populated from input-level data. The following two instances are therefore optional, depending on the scenario.

- Planetary default data
- Gravity default data

```
163 ##include "environment/ephemerides/verif/SIM_prop_planet/data/include/saturn_planet.hh
    h"
164 ##include "environment/ephemerides/verif/SIM_prop_planet/data/include/saturn_sphericaa
    l_gravity.hh"
...
169    jeod::Planet_saturn_default_data saturn_planet_default_data;
170    jeod::SphericalHarmonicsGravitySource_saturn_spherical_default_data
171       saturn_gravity_default_data;
...
244       ("default_data") planet_default_data.initialize ( &planet );
245       ("default_data") gravity_default_data.initialize ( &gravity_source );
```

Note – in this simulation, the Saturn-specific data is passed into the generic propagated-planet simulation-object at the construction of the Saturn-specific instance of the propagated-planet simulation-object.

I.3.2.b Vehicle-based Components

From the vehicle simulation-object (e.g. *vehicle_basic.sm)* our new simulation-object needs:

- A dynamic body. Instead of the *DynBody* that we see in *vehicle_basic.sm*, we need a *PropagatedPlanet*.
- Gravity Controls. This provides the controls over the gravitational fields of other bodies, i.e. those fields through which this planet moves.
- Body actions for initialization of the mass and state of the planet.

```
202 ##include "environment/ephemerides/propagated_planet/include/propagated_planet.hh"
...
204 ##include "environment/gravity/include/spherical_harmonics_gravity_controls.hh"
205 ##include "dynamics/body_action/include/mass_body_init.hh"
206 ##include "dynamics/body_action/include/dyn_body_init_trans_state.hh"
207 ##include "dynamics/body_action/include/dyn_body_init_rot_state.hh"
...
```

```
(continued)
218    jeod::PropagatedPlanet prop_planet;
...
220    jeod::SphericalHarmonicsGravityControls gravity_controls[9];
...
223    jeod::MassBodyInit mass_init;
224    jeod::DynBodyInitTransState trans_init;
225    jeod::DynBodyInitRotState rot_init;
```

Then make the associated initialization and update calls.

```
250       P_EPH  ("initialization") prop_planet.initialize_model (
251            time_manager, dyn_manager);
```

Note - while a *DynBody initialize_model* method takes only a single argument for the Dynamics Manager, a *PropagatedPlanet* equivalent method requires both the Dynamics Manager and Time Manager.

I.3.3 Input File

The propagated planet must be named, and the name of its parent must be specified.

models/environment/ephemerides/verif/SIM_prop_planet/Modified_data/saturn.py

```
 19   saturn.prop_planet.planet_name = "Saturn"
 20   saturn.prop_planet.parent_name = "SSBary.inertial"
```

Then the mass-initialization, state-initializations, and gravity controls set just as for any other vehicle.

The propagated planet has aspects of a planet with a planet-centered-inertial reference-frame, and that of a dynamic-body with a composite-body reference-frame. These two reference-frames are tied together, almost like a master-slave implementation. It is important to specify from which reference-frame the state is obtained. The translational and rotational state are considered independently; consequently there are four options for the value *commanded_mode*

```
PropagatedPlanet::TransFromPlanet_RotFromPlanet
PropagatedPlanet::TransFromPlanet_RotFromBody
PropagatedPlanet::TransFromBody_RotFromPlanet
PropagatedPlanet::TransFromBody_RotFromBody
```

I.3.4 Logging Data

The state of the planet's planet-centered-inertial reference-frame and/or that of the propagated-planet's composite-body frame may be logged.

```
dr_group.add_variable("asteroid.prop_planet.body.composite_body.state.trans.position["+str(ii)+"]" )
dr_group.add_variable("asteroid.planet.inertial.state.trans.position[" + str(ii) + "]" )
```

I.4 Advanced Integration

Document reference: *utils/integration/docs/integration.pdf*

The Integration Model works behind the scenes, propagating – among other things – the rotational and translational states of vehicles based on the forces and torques acting on them. It uses numerical integration to solve the equations of motion. JEOD supports a broad selection of integration algorithms; for a full list of these algorithms, see section B.4. The Integration Model document provides more detailed descriptions of a subset of the available algorithms.

The basics of setting up the model to handle the state of a vehicle were presented in section B.4 where the Runge-Kutta 4th order integrator implementation was described as an illustrative example. In this section, we briefly consider the more advanced capabilities of the integration model, including some new concepts:

- The integrable object is a variable of some nature that may be integrated
- The integration group is a group of integrable objects, all integrated in one block
- The integration loop is the connection of all jobs associated with integrating an integration group.

I.4.1 Integrable Object

An integrable object is an ER7-Utilities abstract concept that “wraps” some variable whose value may be integrated, adding to it things like an integration group to which it belongs, and integration controls for how it should be integrated. The *IntegrableObject* class itself is abstract; it cannot be instantiated. The intent is to make specific *IntegrableObject* types, each with specific rules for manipulating (creating/destroying/resetting) integrators, and for using those integrators to advance the state of the object itself.

Thus far, our exposure to *IntegrableObject* has not been explicitly called out, but we have been using them throughout this course. A *DynBody* is an *IntegrableObject*. The *DynBody* provides all of the necessary methods to make the *IntegrableObject* base into an instantiable class capable of integrating the 6-dof state of a dynamic body.

Additional *IntegrableObject* types might be a simple scalar quantity, or a 3-vector. See, for example, the *ThermalIntegrableObject* in *interactions*/*thermal_rider*, or see the Contributions area on the JEOD-wiki.

I.4.2 Integration Group

A simplistic description of an *IntegrationGroup* is some entity that houses a collection of *IntegrableObject* items. All *IntegrableObjects* in the *IntegrationGroup* will be integrated at the same frequency, using the same integration algorithm. But different groups may be integrating at different rates or with different algorithms. The groups can be independent of one another.

Thus far, our exposure to *IntegrationGroups* has been lurking below the surface. The default implementation of the Dynamics Manager provides a single group, so we have had no need to investigate the management of that group; the Dynamics Manager has been doing it for us. Now we will consider the implications of moving to an architecture involving multiple Integration Groups.

In the single-group paradigm, we have a single IntegLoop (see next section for details on integration-loops) to which we add the dynamics simulation-object, and any simulation-object containing a derivative-class job.

In the multiple-group paradigm, we have multiple integration loops, one per group. Some of the previous responsibilities of the dynamics simulation-object now get re-assigned. Most notably, the derivative-class jobs (like gravitation) and the only integration-class job are moved to a new integration-loop simulation-object whose responsibilities are exclusively managing the integration for the group.

Any simulation may include several of these new integration-loop simulation-objects, but there remains a single (albeit slimmed-down) dynamics simulation-object. This dynamics simulation-object DOES NOT get added to ANY of the integration loops, it **manages** the integration-loops.

Simulation-objects that contain either a derivative class job OR an integrable object (e.g. a *DynBody*, a vehicle) must each be assigned to **one** (and only one) group. All integrable objects within the same simulation-object SHOULD belong to the same group; this is handled automatically behind the scenes for JEOD-defined integrable objects. Therefore, all JEOD-defined integrable-objects within any given simulation-object SHALL be integrated using the same techniques and at the same rate.

Note that for simple simulations (the default behavior), for which there is only 1 group in the simulation, all objects in the simulation will be integrated in the same manner. In this single-group paradigm, all of the integration is processed through the Dynamics Manager's *integrate* method.

In the case of multiple groups, the Dynamics Manager keeps track of the groups, processing each group's *integrate* method one at a time according to its schedule. The individual groups manage the integration of the entities in their respective simulation-objects.

I.4.3 IntegrationLoop

An *IntegrationLoop* is a Trick construct that forms job queues for collections of Trick simulation-objects. For every Trick simulation object assigned to an *IntegrationLoop*, the *IntegrationLoop* will schedule those jobs tagged with one of the following classifications:

- pre-integration

- derivative
- integration
- dynamic-events
- post-integration

Consequently, the boundaries of the Trick simulation-objects define which methods get assigned to which integration-loop.

Recognize here the potential for erroneous implementation.

- Each integration-loop will also be associated with an integration-group (of integrable-objects).
- All integrable-objects are ultimately associated with an integration-loop via the integration-group in which they are placed.
- The integration-loop grabs all of the jobs tagged with one of those 5 classifications from its associated simulation-objects, and executes those according to the schedule defined by the execution rate of the integration-loop.
- So with the execution of an integration-loop, all of the associated jobs will be executed, and all of the integrable-objects in the associated integration-group will be integrated.
- If not managed properly, an integrable-object could be moved into a group that is associated with an integration-loop that is not the integration-loop that is managing the derivative-class jobs (for example) for that integrable-object. This could result in those derivative-class jobs being called in a sequence that is inconsistent with the integration of the integrable-object.

Consequently, it is strongly recommended – especially for integrable objects that influence or are influenced by one of the 5 classifications of jobs listed above – that each integrable-object be assigned to the integration-group that is associated with the integration-loop that is managing that integrable-object's simulation-object.

From within JEOD, the only integrable-objects are *DynBody* instances, and there is code that manages multiple groups and loops in such a way that the user is never responsible for directly assigning integrable-objects to an integration-group. Instead, simulation-objects get assigned to integration-loops, and – behind the scenes – the integrable-objects within those simulation-objects are identified and assigned to the associated groups.

However, all of these classes are extensible, and there is the potential for model developers to override this safety feature.

> As an illustrative example of this complex reasoning, consider the following proto-code based S_define:

```
class TestSimObject {
 IntegrableObject object1;
 IntegrableObject object2;
};
TestSimObject test;

JeodIntegLoopSimObject loop1 (...,
                              test,
                              ...);
JeodIntegLoopSimObject loop2 (...);
```

In this example, the *test* simulation-object is added to integration-loop *loop1*. When *loop1* is created, an integration-group (let's call it *group1*) is also created automatically. It is strongly recommended that o*bject1* and *object2* be assigned to *group1*. If *object1* and *object2* are JEOD-defined integrable-objects, this will be handled automatically and cannot be broken. If one or both is independent of JEOD, the responsibility falls to the user to ensure that they are added to the correct group. Or else ...!!

I.4.4 Setting up Multiple Integration Groups

The first step is to include the default simulation-module, *integ_loop.sm*. Like the *jeod_sys.sm* and *trick_sys.sm* files, this should not be edited.

```
Example
#include "JEOD_S_modules/integ_loop.sm"
```

Next, instead of using the simulation-module *dynamics.sm*, include *dynamics_multi_group.sm*. This may be edited as necessary.

```
#include "JEOD_S_modules/dynamics_multi_group.sm
```

It is necessary to instantiate an instance of *IntegrationGroup* (or *DynamicsIntegrationGroup*, a derivative of *IntegrationGroup* intended for dyn-bodies), and constructors for any integration methods you plan to use. Depending on the complexity of the desired integration, these may be added to an existing simulation-object (e.g. *DynamicsSimObject*), or it may be preferable to create a new simulation-object, such as *IntegrationSimObject* (or any other meaningful name). Remember also to add headers for these instantiations.

Remember to preface the constructor types with *"er7_utils::"* to properly namespace these instances.

```
##include "dynamics/dyn_manager/include/dynamics_integration_group.hh"
##include "er7_utils/integration/abm4/include/abm4_integrator_constructor.hh"
##include "er7_utils/integration/beeman/include/beeman_integrator_constructor.hh"
etc.
class IntegrationSimObject: public Trick::SimObject {
...
jeod::DynamicsIntegrationGroup group;
er7_utils::ABM4IntegratorConstructor abm4_integ_constructor;
er7_utils::BeemanIntegratorConstructor beeman_integ_constructor;
```

Finally (and this part is a little awkward), you must create pointers to the integrator constructors, of generic type er7_utils::IntegratorContructor. In the initializer list (part of the sim-object constructor), these must be populated with the addresses of the integrator constructors just instantiated

```
er7_utils::IntegratorConstructor * abm4_ptr;
er7_utils::IntegratorConstructor * beeman_ptr;
...
IntegrationSimObject()
 :
 abm4_ptr( & abm4_integ_constructor),
 beeman_ptr( &beeman_integ_constructor)
...
};
IntegrationSimObject integ_object;
```

At the end of the S_define file, make as many integration loops as desired in place of the single-loop integration statement:

Replace

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

with

```
JeodIntegLoopSimObject name_of_group_1 (...)
JeodIntegLoopSimObject name_of_group_2 (...)
JeodIntegLoopSimObject name_of_group_3 (...)
etc.
```

The constructor arguments (shown as ... above) are as follows:

1. The integration-rate. With a pre-defined value (e.g. DYNAMICS), only the name (DYNAMICS) is needed.

2. The declared pointers to the instances of the desired *IntegratorConstructor.* For example, to use an Euler integration, declare an instance of *EulerIntegratorConstructor* in the *integ_object* simulation-object, and put the name of that instantiation here. In the example above, a *BeemanIntegratorConstructor* called *beeman_integ_constructor* was instantiated in the simulation-object *integ_object*. To use the Beeman integrator with this group, argument #2 would be *integ_object.beeman_integ_constructor.*

   NOTE – do not just pass in the address of the integrator-constructor; this pointer must have longevity, and a pointer created by passing an address will be destroyed as soon as the method returns. You must pass a permanently instantiated pointer.

3. An instance of the *IntegrationGroup* class (e.g. *integ_object.group*). Note that this one instance can be used in every integration group, you do NOT need to instantiate one per group.

4. The *TimeManager* object for the simulation. There should be only one; if using the standard sim-modules, it will be *jeod_time.time_manager.*

5. The Dynamics Manager *(DynManager)* object. Again, there should be only one in the simulation; if using the standard sim-modules, it will be *dynamics.dyn_manager*.

6. The gravity manager *(GravityManager)* object. Again, there should be only one in the simulation; if using the standard sim-modules, it will be *env.gravity_manager*.

7. The address of the first simulation-object to be integrated in the group. For example, with a simulation-object called *vehicle_1*, argument #7 will be &*vehicle_1*.

8. Additional addresses for other simulation-objects to be integrated in this group.

9. Finish the argument with NULL to indicate that all desired simulation-objects have been entered.

Note – each such call automatically constructs a *JeodDynbodyIntegrationLoop* and adds itself (the instance of *JeodIntegLoopSimObject*) into the loop.

Example:

```
JeodIntegLoopSimObject fast_dynamics (0.1,
                                      integ_object.rk4_ptr,
                                      integ_object.group,
                                      jeod_time.time_manager,
                                      dynamics.dyn_manager,
                                      env.gravity_manager,
                                      &vehicle1,
                                      NULL);
JeodIntegLoopSimObject slow_environment (10.0,
                                      integ_object.rk4_ptr,
                                      integ_object.group,
                                      jeod_time.time_manager,
                                      dynamics.dyn_manager,
                                      env.gravity_manager,
                                      &vehicle2,
                                      &vehicle3,
                                      NULL);
```

NOTE – integration loops may be created and left empty for later population.

I.4.4.a Behind the Scenes (optional)

Each *JeodIntegLoopSimObject* contains *integ_loop,* an instance of *JeodDynbodyIntegrationLoop (see utils/sim_interface/include/trick_dynbody_integ_loop.hh)*, which is a derivative class of Trick's *IntegLoopScheduler* class.

At construction of each *JeodIntegLoopSimObject* instance, *integ_loop* is constructed, and then all of the simulation-objects specified in the constructor argument list get added to *integ_loop.* Because *integ_loop* is an instance of *IntegLoopScheduler* (by inheritance), it now becomes responsible for queuing the 5 classifications of methods associated with integration that are present in all of those simulation-objects.

At initialization, the *initialize_integ_loop( )* method is called with most-urgent priority (*P1*). This creates an integration-group and adds it to the Dynamics Manager. The Dynamics Manager will now manage the integration of the elements of this group.

Also at initialization, the *DynamicsManager*::*initialize_simulation( )* method is called (from either *dynamics.sm* or *dynamics_multi_group.sm*, or a user-specified dynamics.sm file). This method includes a call to *initialize_integ_groups( )*, which processes the list of integration-groups (including those added by the *initialize_integ_loop( )* method above), calling *DynamicsIntegrationGroup::initialize_group* on each one.

This call propagates up into the integration-loop, which processes the Dynamics Manager's list of *DynBody* instances and adds them to the integration-group.

I.4.5 Managing non-JEOD Integrable Objects

At simulation initialization, all integrable-objects that are known to JEOD – namely *DynBody* instances – will get added to the integration groups associated with the integration-loops that are managing their respective simulation-objects.

It is not possible to anticipate all possible types of integrable-objects. Indeed it is completely unreasonable to require that all user-defined integrable-objects be registered with the Dynamics Manager. Consequently, any user-defined integrable-objects will not be automatically known to the Dynamics Manager (unless the developer has added code to make them known). Consequently, the Dynamics Manager cannot be responsible for adding these integrable-objects to the associated integration-groups.

However, the DynBody class provides one little extra – a list of (actually an STL-vector of pointers to) integrable objects that are associated with the DynBody instance. Integrable-objects may be added to this collection, *associated_integrable_objects*, with the *add_integrable_object(er7_utils::IntegrableObject &)* method:

```
Example from contribs – Vector3IntegrableObject:
139  dyn_body->add_integrable_object( *this);
```

If this method is called *before* initialization of the *DynBody* instance, then this new integrable-object will automatically be added to the integration group alongside the *DynBody* instance.

Otherwise, users are responsible for adding their own integrable objects to the respective integration groups, using such as:

```
134  integ_group->add_integrable_object( *this)
```

Users are responsible for adding user-defined integrable-objects to the appropriate integration-groups at simulation initialization.

I.4.6 Moving a Simulation Object to Another Loop

In some situations, it is desirable to move a simulation-object from one integration-loop to another. For example, consider a vehicle departing an Earth orbit onto a free-coasting interplanetary trajectory; in this case it may be desirable to switch the integrator from, say, an RK4 integrator with an integration step of 1.0 seconds to, say, a Gauss-Jackson integrator with an integration step of 1.0 hours. This transition is straightforward, with a command structure of the form:

<loop-name>.integ_loop.add_sim_object(<sim-object>)

where *<loop-name>* is the instance of *JeodIntegLoopSimObject* TO which the object is moving, and *<sim-object>* is the simulation-object whose integrable objects are being moved.

For example, continuing with the example *S_define* above, to move the *vehicle2* sim-object (which contains the *DynBody IntegrableObject*) from the slow_environment group (where it is originally assigned) to the fast_dynamics group:

```
fast_dynamics.integ_loop.add_sim_object(vehicle2)
```

This *add_sim_object* command can be executed via the code-base, or entered at the input-file level. It is typically executed in response to some dynamic event or the crossing of some threshold. Note that there is no need to *remove* the object from its old group, that is done automatically. An object can only be in one group.

When *add_sim_object(...)* gets called at runtime, the *IntegLoopScheduler* components of *integ_loop* will move the simulation-object and re-schedule all integration-related jobs (see the 5 job-type classifications) from that simulation-object.

In addition, the *JeodDynbodyIntegrationLoop* additions to *IntegLoopScheduler* checks all instances of *DynBody* known to the Dynamics Manager. Any that are found to be in the moving simulation-object are moved to the new integration-group through a cascade of method calls:

```
utils/sim_interface/src/trickdynbody_integ_loop.cc
275 JeodDynBodyIntegLoop::add_sim_object(
...
289      add_sim_object_bodies (sim_obj);
...

319 JeodDynbodyIntegrationLoop::add_sim_object_bodies (
...
330         integ_group->add_dyn_body (*body);
...
```

```
dynamics/dyn_manager/src/dynamics_integration_group.cc
177 DynamicsIntegrationGroup::add_dyn_body (
...
206   dyn_body.set_integration_group (*this);
```

Furthermore, **if the DynamicsIntegrationGroup flag *bodies_integrated_separately* has not been turned off**, (it defaults to true, and indicates whether the *DynBody* instances are to be integrated via the *DynBody::integrate()* command), then all of the integrable objects associated with this *DynBody* will also be moved to the new group.

```
210   if ((integ_controls != NULL) && bodies_integrated_separately) {
...
220      for (std::vector<er7_utils::IntegrableObject *>::const_iterator it =
221              associated_objects.begin();
222           it != associated_objects.end();
223           ++it) {
224         add_integrable_object(**it);
```

Note that if the user's integrable objects are not associated with a DynBody, then the user is responsible for moving them to their new integration group when the respective simulation-object is switched to a new loop.

WARNINGS!

Remember that only those integrable-objects known to the Dynamics Manager (i.e. *DynBody* instances and associated integrable objects) will be moved automatically.

The integration for one group will be processed for the entire of its integration step before the next group starts. Where both groups have the same integration step, then both groups always finish their integration at the same place in time. However, when the integration rates differ, the completion of one group will often represent a different time than the other group.

Thus, care should be taken when moving between groups with different integration rates. Continuing with the same example, *fast_dynamics* runs every 0.1 seconds, while *slow_environment* runs only every 10 seconds. Objects in the two groups will be integrated to the same epoch in time only once every 10 seconds (lowest common multiple of the two rates). Suppose the *vehicle2* object was moved at an intermediate time, say at t=24.3 seconds. Now, the *vehicle2* object only has a known "state" at t=20.0 or t=30.0 seconds. Moving it to *fast_dynamics* at t=24.3 seconds will result in either:

- if it has not yet been integrated to 30.0s, the period 20.0-24.3 seconds will not be integrated at all, and the effect on the state during that time will be lost.
- if it has already been integrated to 30.0 seconds, that state will be carried across, and re-integrated during the period 243.3-30.0 seconds. The effect on the state during this period will be counted twice.

Only move objects between groups, and simulation-objects between integration-loops, at a time that represents the end of an integration-step for BOTH groups/loops.

I.4.7 Adding / Removing Integrable Objects Directly

Integrable objects can be added and removed from an integration group directly. It is more typical that users would have direct access to an integration loop rather than the group, so pass-through methods are made available from the Integration Loop. It is also possible, usually from within code rather than from the user-interface, that a user may want to manipulate the group directly. Both options are presented here.

I.4.7.a Manipulating the Integration Loop

The *JeodDynBodyIntegrationLoop* class handles the management of those integrable objects that JEOD knows about – namely, *DynBody* instances and integrable objects associated with *DynBody* instances. But it cannot anticipate other types of integrable-objects. To facilitate direct addition and removal of integrable-objects to/from their associated integration group, public methods are provided; both methods take a single reference to an integrable-object:

- *JeodDynBodyIntegrationLoop::add_integrable_object(er7_utils::IntegrableObject &)*
- *JeodDynBodyIntegrationLoop::remove_integrable_object(er7_utils::IntegrableObject &)*

Attempts to add *DynBody* instances through this mechanism will fail because that must be handled via the *add_sim_object( )* call. Users are responsible for ensuring that integrable-objects that are both non-*DynBody* and not associated with a *DynBody* are placed in the correct integration group.

There is no such protection on the remove capability; users are permitted to remove any integrable objects from a n integration loop (and thereby from its integration group). But recognize that in the case of *DynBody*

instances, a more accessible method would be to turn off the *translational_dynamics* and *rotational_dynamics* flags.

The removal of the last integration object from a group has no extraneous effect. The loop/group will still be available for later re-population if desired.

I.4.7.b Manipulating the Integration Group

The JeodDynBodyIntegrationLoop creates an instance of DynamicsINtegrationGroup, which inherits from JeodIntegrationGroup. This latter type proves the methods:

- *JeodIntegrationGroup::add_integrable_object(er7_utils::IntegrableObject &)*
- *JeodIntegrationGroup::remove_integrable_object(er7_utils::IntegrableObject &)*

These are the methods that are called if the user goes through the Integration Loop methods. The preferred practice is to go through the Integration Loop mechanism, especially for operation from the user-interface. These methods have no protection against misuse; use at your own risk.

Exercise 41. Multiple Integration Groups

I.5 Mathematical Tools

Document reference: *utils/math/docs/math.pdf*

The Math Model provides a selection of mathematical functions, including vector and matrix manipulation tools. It comprises three classes of operators: *Matrix3x3*, *Vector3*, and *Numerical*.

The Math functions, in general, are rarely used directly in the S_define. The model is provided as a utility for within many of the JEOD models, as well as for further development. To use these capabilities in additional code development work, insert the includes that you need at the top of the source file:

```
Example
#include ``utils/math/include/matrix3x3.hh''
#include ``utils/math/include/vector3.hh''
#include ``utils/math/include/numerical.hh''
```

Note that most of the functions are defined in the header files *matrix3x3_inline.hh*, *vector3_inline.hh*, and *numerical_inline.hh*, which are included automatically from the main header files.

Some functions (typically the *Vector3* and *Numerical* class methods) allow for chaining of operations, while others (typically the Matrix3x3 class methods) are intended to be stand-alone methods.

```
Examples of chained methods
double square_of_hypoteneuse = Vector3::vmagsq(vect_a) + Vector3::vmagsq(vect_b)
double cos_theta = Vector3::dot(Vector3::normalize(vect_a),Vector3::normalize(vect_b))
```

```
Example of stand-alone method
Matrix3x3::transpose(original_matrix, transposed_matrix)
```

Appendix D includes a list of frequently-used methods from all three classes within the Math Model.

I.6 Quaternion

Document reference: *utils/quaternion/docs/quaternion.pdf*

Quaternions are a 4-vector; in many discussions (and in the JEOD-code) a quaternion is broken into 2 components: a scalar component *(scalar* in the Quaternion class) and a 3-vector (*vector* in the Quaternion class). The Quaternion Model document provides detailed coverage of quaternion arithmetic operations.

In JEOD, all orientation-states are provided using both a transformation matrix and a quaternion to represent the rotation from the reference reference-frame to the current reference-frame. The Quaternion Model provides methods to allow the easy manipulation of quaternions, including:

- Quaternion normalization
- Quaternion multiplication
- Transformation matrix ~ Quaternion conversions
- Quaternion representations of Eigen rotation, and vice-versa.

In general, when a quaternion is used to represent a relative orientation represented by a rotation through angle θ about some axis $\hat{u}$, its value can be expressed as

$$\boldsymbol{Q} = \begin{bmatrix} \cos\frac{\theta}{2} \\ \sin\frac{\theta}{2}\hat{u} \end{bmatrix}$$

Appendix D includes a list of frequently-used quaternion methods.

WARNINGS

Use unit-quaternions for orientation

Quaternions have a magnitude, analogous to that of a vector. Quaternions used to represent an orientation must be unit-quaternions (have a magnitude of value 1), just as vectors used to represent a pure direction must be unit vectors. Not all quaternions are unit-quaternions. When interfacing with other models, ensure that the incoming quaternions are either unit-quaternions, or normalize them to be unit-quaternions if they are not.

Left-quaternions and right-quaternions

Quaternions can be expressed in two DIFFERENT WAYS, often referred to as left-quaternions and right-quaternions. **JEOD uses left-quaternions.**

The difference between left and right quaternions is a simple concept. When a quaternion is used to represent a relative orientation, that means that it can be used to rotate a vector, *a*, to a vector, *b*.

In doing so, we make *a* into a quaternion-form and multiply by the quaternion, $\boldsymbol{Q}$, and its conjugate $\boldsymbol{Q}^*$ to give a new quaternion-form that includes vector *b* in one of two ways.

$$\begin{bmatrix} 0 \\ \vec{b} \end{bmatrix} = \boldsymbol{Q} \begin{bmatrix} 0 \\ \vec{a} \end{bmatrix} \boldsymbol{Q}^* \qquad\qquad \begin{bmatrix} 0 \\ \vec{b} \end{bmatrix} = \boldsymbol{Q}^* \begin{bmatrix} 0 \\ \vec{a} \end{bmatrix} \boldsymbol{Q}$$

In the first case, the quaternion, $\boldsymbol{Q}$, is called a left-quaternion, because it is applied on the left of the original vector. In the second case, the quaternion is described as a right-quaternion, because it is applied to the right of the original vector.

Consequently, the left-quaternion - here, we are limiting our discussion to the unit quaternions that we use to represent relative attitudes - is the conjugate of the right-quaternion (more generally, the left quaternion is the inverse of the right quaternion - and vice-versa - but the conjugate and inverse of a unit quaternion are equivalent).

For a quaternion $\boldsymbol{Q} = \begin{bmatrix} q_s \\ \vec{q}_v \end{bmatrix}$, the conjugate, $\boldsymbol{Q}^* = \begin{bmatrix} q_s \\ -\vec{q}_v \end{bmatrix}$

where q_s and $\vec{q}_v$ are the scalar and vector components respectively of the quaternion, $\boldsymbol{Q}$.

Quaternion representations

While JEOD, and many other applications, represent quaternions in a “1+3” scalar+vector format, other applications use a 4-vector. Usually, the scalar component is represented by the first element in the 4-vector, but sometimes it is placed at the other end of the 4-vector.

When interfacing with other models, users must be aware how the term *quaternion* is being interpreted in those models. JEOD provides numerous methods to assist with importing quaternions from other models, whether they are left- or right- quaternions, expressed as a 4-vector or scalar + 3-vector, or other inconsistency with the JEOD interpretation. For example:

- to normalize a non-unit quaternion into a unit-quaternion:

  ```
  quaternion.normalize();
  ```

  just changes *quaternion,* while

  ```
  quaternion.normalize(unit_quaternion);
  ```

  makes *unit_quaternion* into the normalized version of *quaternion.*

- to convert from a unit right-quaternion to the unit left-quaternion, use the Quaternion::conjugate method:

  ```
  quaternion.conjugate();
  ```

  just changes *quaternion,* while

  ```
  right_quaternion.conjugate(left_quaternion)
  ```

  leaves the original (*right_quaternion*) intact, and sets *left_quaternion* only.

- If the quaternions are expressed as a 4-vector, with scalar component first,

  ```
  quaternion.copy_from(incoming_quat)
  ```

  will take the 4-vector and convert it into the "scalar + 3-vector" form that JEOD uses, then

  ```
  quaternion.copy_to(incoming_quat)
  ```

  will copy it back to a 4-vector (scalar first) for exporting again.

See Appendix D for more quaternion methods.

I.7 Message Handler

Document reference: *utils/message/docs/message.pdf*

The Message-Handler Model provides the framework for communication between the simulation and the user. It is automatically included with the specific implementation of the *JeodSimInterface*, which is a necessary component for all JEOD simulations. Error message, warnings, unusual behavior, etc. all get sent through the Message Handler. There are five primary levels of error:

1. Failure. The simulation will stop
2. Error. Errors almost always invalidate the simulation output, but do not cause the simulation to terminate. If any unexpected Error messages are observed, it is safe to conclude that the simulation has problems.
3. Warning. Warnings are sent when a model is being used in a manner in which it was not intended. This does not necessarily mean that the usage is inappropriate, but developers writing simulations that produce warnings should probably have good reason for using the architecture in that way. Reporting of warnings is on by default but may be disabled.
4. Notice. Notices are primarily informational messages – for example a model has performed some unusual or unexpected task that was anticipated during development but for very limited situations. Notices are not usually reported, but may be enabled.
5. Debug. A very low priority message used for debugging purposes and often removed from the code base thereafter. This level is primarily used during code development (as opposed to simulation development) only. Reporting is disabled by default, but may be enabled.

The *MessageHandler* itself must be extended to use a certain communication path; for the purposes of this course, we are using Trick simulations, so will use the Trick-appropriate paths, as specified in the sub-class, *TrickMessageHandler*, which is included in the Trick-specific sim interface, *JeodTrickSimInterface*.

I.7.1 Setting Message Handler Parameters

I.7.1.a Suppressing Messages based on Importance

The extent to which messages are written to the appropriate output is a user-defined setting. Messages will be suppressed if they have a tag-value that is greater than the value *suppression_level*. Note that more important messages have lower-valued tags. The default setting is such that messages at the Notice and Debug levels are suppressed. The tag-values for the different message types are:

- 999: Debug
- 99: Notice
- 9: Warning
- 0: Error
- -1: Fail

Note that messages at the Fail level cannot be suppressed.

To change the suppression level, call the *set_suppression_level* method. This may be applied with a number:

```
Example
jeod_sys.message_handler.set_suppression_level(10)
```

or an enumerated value:

```
Example
jeod_sys.message_handler.set_suppression_level(trick.MessageHandler::Warning)
```

I.7.1.b Setting Message Contents

There are two flags that can be set to suppress certain components of an error message. By default, all messages provide:

- the message severity and type,
- the file-name and line number where the instance took place,
- a textual summary of the incident.

Setting the *suppress_id* flag prevents the message type and severity from being output.

Setting the *suppress_location* flag prevents the filename and line number from being output.

```
Example
jeod_sys.message_handler.set_suppress_id(true)
jeod_sys.message_handler.set_suppress_location(true)
```

Note that no component of messages at the Fail level may be suppressed.

I.7.2 Using the Message Handler with New Models

This process is explained in the model documentation.

Chapter J Miscellaneous Exercises

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

Appendix A Introduction to Trick

This is not intended to be a definitive lesson in using Trick, it only covers the essentials for using JEOD. The Trick team produces their own tutorial and training materials.

There are several types of files that we will be working with:

A.1 S_define

The S_define file defines the simulation. It provides the structure, the sequencing for all of the function calls. To make the S_define more easily navigable and manageable, the simulation typically comprises a collection of simulation objects, each defined as a C++ class and instantiated separately. The definition of each simulation object may be done in the S_define file itself, or it may be performed in another file and included from the S_define file.

The separation of the overall simulation into simulation objects is something of an art. Ideally, the objects should be as self-contained as possible, with minimal necessity for any methods called from one simulation object to know of data or methods defined in another. Simulation objects that are too small are likely to need significant data elements from elsewhere. Simulation objects that are allowed to grow too large become difficult to maintain.

In JEOD, we typically have one object for maintaining time, one for managing the dynamics and integrating everything together, one for each vehicle, one for each planet, and one for relative states. For some scenarios, it is sometimes also desirable to separate out the vehicle-environment interactions into another simulation object. Where one simulation-object needs data from another simulation-object (this is inevitable if the simulation is not to be written in one object), we typically use references, created at construction time, to the necessary data. See section A.4.4 for a guide to this process. Simple Guidance, Navigation and/or Controls algorithms may be included within the sim-object of the associated vehicle. For simulations involving extensive algorithm implementation or full flight software, the architecture may be improved by moving these to an independent simulation-object. The trick is to generate an architecture that provides a convenient structure of mostly-self-contained building blocks; too small and the isimulation becomes difficult to navigate, too large and the blocks become unwieldy and difficult to manage.

The following example provides a fairly simple S_define, annotated by section to help beginners identify the components of an S_define.

First, there is an optional heading. Notice that all of these lines begin with “//”. That indicates that the contents of the line are comments only, and will be ignored by the compiler.

```
//==========================TRICK HEADER====================
// PURPOSE:
//=========================================================================
// This simulation provides an example of a simple of JEOD simulation,
// and makes use of the available JEOD S_modules. The purpose it to illistrate
// the use of scheduled class jobs, call intervals, and default data jobs.
//
//          sys - Trick runtime executive and data recording routines
//         time - Universal time
//     dynamics - Orbital dynamics
//     vehicle - Vehicle dynamics model
//
//=========================================================================
```

Next, we typically define values that are going to be used within the S_define; rather than enter the value every time, we sign the value to some variable, and use the variable. This is most often done to define the rate at which functions are called. If we want to change the rate, it is easier to change the value here rather than changing it at every function call. In this case, we have two values, one providing a rate called DYNAMICS, being called every 0.5 ticks of *sim-time*, and one called CALENDAR_INTERVAL, being called every 1.0 ticks of *sim-time* (see section B.2.1 for discussion of *sim-time*).

```
// Define job calling intervals
#define DYNAMICS       0.5
#define CALENDAR_INTERVAL 1.0
```

Next come the default simulation-modules. All Trick simulations require *trick_sys*, and all simulations that bring in JEOD require *jeod_sys*. Any simulations using the additional (i.e. excepting *jeod_sys*) default simulation modules that are provided with JEOD should also include the *default_priority_settings* module.

```
 // Include the default system classes:
#include "sim_objects/default_trick_sys.sm"
#include "JEOD_S_modules/jeod_sys.sm"

#include "JEOD_S_modules/default_priority_settings.sm"
```

Next, we have the first simulation-object. For most JEOD simulation, this typically defines time. We optionally start with a brief heading to explain what the simulation-object represents, then include all of the headers for all objects instantiated within the simulation-object.

Notes:

1. If an object has already been instantiated in a previous simulation-object, its header does not need to be included again, but should be anyway to assist with later code maintenance. Duplicate includes do not cause a problem, and having it there saves poring through previous simulation-objects looking for it.

2. The include syntax really is *##include* (double-pound/hash) to distinguish these includes from *files* that are going to be included (single pound/hash, *#include*), such as the S_modules included in the previous step.

```
//===========================================================================
// Create the time class
// Includes dyn_time and two clocks; tai and ut1.
//===========================================================================
// Include headers for classes that this class contains:
##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_converter_dyn_tai.hh"

##include "environment/time/include/time_ut1.hh"
##include "environment/time/include/time_converter_tai_ut1.hh"
##include "environment/time/data/include/tai_to_ut1.hh"
```

Next the class is defined. We start with the name of the class type, which always inherits from the SimObject class defined in Trick.

```
class JeodTimeSimObject: public Trick::SimObject {
```

Typically, the data elements of a simulation-object are declared as being public – that is, they can be accessed from outside the class. This is especially important for any variables that are used as inputs to the simulation, or as outputs from the simulation. Internal variables may be protected. References to other variables in other simulation-objects must be protected.

With thepublic/protected/private decisions made, the data elements can be instantiated. Notice in this example that there are seven data elements, to go with the seven *##include* statements above. While it helps maintainability to have the header files and data elements in the same order, it is not necessary, and the JEOD classes are all identified with names that match the filenames where they are defined.

```
public:

   // Member data

   // The time manager and its initializer
   jeod::TimeManager     time_manager;
   jeod::TimeManagerInit time_manager_init;

   // TAI and dynamic time to TAI converter
   jeod::TimeTAI               time_tai;
   jeod::TimeConverter_Dyn_TAI time_converter_dyn_tai;

   // UT1, TAI to UT1 converter, and UT1 default data
   jeod::TimeUT1                                       time_ut1;
   jeod::TimeConverter_TAI_UT1                         time_converter_tai_ut1;
   jeod::TimeConverter_TAI_UT1_tai_to_ut1_default_data time_converter_tai_ut1_default_data;
```

Next, the class constructor is defined. Here, any data from other simulation-objects may be passed in for storage as references (preferred) or pointers (acceptable). In this case, there are none, but the next simulation-object does have these.

```
   // Constructor
   JeodTimeSimObject () {
```

The default data jobs (tagged “default_data”) are often defined first because they are the first processed, but this is by convention only and not required. In this case, there is one, and it is embedded within the initialization jobs. There may be no default data jobs within a simulation-object.

The initialization-class jobs (tagged “initialization”) are often defined next. They may be given a priority, in this case *P_TIME*, that must be defined beforehand. In this case, the definition of *P_TIME* is made in the *default_priority_settings* file included earlier.

The scheduled class jobs are given a calling frequency. In this case, there are two:

1. *time_manager.update* is called at the *DYNAMICS* rate (0.5 ticks of *sim-time*), and
2. *time_ut1.calendar_update* is called at the *CALENDAR_INTERVAL* rate (1.0 ticks of *sim-time*).

Both rates were defined earlier in the file.

The derivative class jobs (tagged with “derivative”) also appear in this section. There are no derivative class jobs in this example simulation-object.

```
      // Default data and initialization jobs
      //
      P_TIME  ("initialization") time_manager.register_type (
          time_tai);
      P_TIME  ("initialization") time_manager.register_converter (
          time_converter_dyn_tai);

      ("default_data") time_converter_tai_ut1_default_data.initialize (
         &time_converter_tai_ut1);
      P_TIME  ("initialization") time_manager.register_type (
          time_ut1);
      P_TIME  ("initialization") time_manager.register_converter (
          time_converter_tai_ut1);
      P_TIME  ("initialization") time_manager.initialize (
          &time_manager_init);
      //
      // Scheduled jobs
      //
      (DYNAMICS, "environment") time_manager.update (
          exec_get_sim_time());

      //
      // Initialization and scheduled jobs for calendar updates
      // The UT1 calendar representation is initialized and updated.
      //
      P_TIME ("initialization") time_ut1.calendar_update (
          exec_get_sim_time());

      (CALENDAR_INTERVAL, "environment") time_ut1.calendar_update (
          exec_get_sim_time());
   }
```

Private members of a class can only be accessed from within the class. In this case, we define the copy-constructor and “operator=” to ensure that they cannot be inadvertently called from elsewhere, and make them empty to ensure that they do not called at all.

```
private:
   JeodTimeSimObject (const JeodTimeSimObject&);
   JeodTimeSimObject & operator = (const JeodTimeSimObject&);
};
```

Next, the simulation-object is instantiated. In this case, there are no arguments (because the constructor was defined with none). The next two simulation-objects we will look at do require arguments to the constructor.

```
JeodTimeSimObject jeod_time;
```

The next two simulation-objects are defined and instantiated within external files, included from the S_define.

```
// Include the most basic dynamics class and object
#include "JEOD_S_modules/dynamics_init_only.sm"


// Include a basic vehicle class and object
#include "JEOD_S_modules/vehicle_basic.sm"
```

There often follows an integration statement.

A.2 Input Data

There are three different types of input data files – default data, modified data, and input file data.

Default Data

Default data is the most “fixed”, and comes in three flavors:

1. At the highest level is the data that is populated by the code constructors when the data elements are first constructed. These include data values athat are necessary for the model to function, as well as default settings that are assumed for verification of appropriate configuration (e.g. setting an enumeration that is intended to be user-assigned to “Undefined” so that its assignment may be checked at initialization).

2. Below this is model-level default-data. These data settings – elements such as large arrays – are typically too involved to be processed in the initializer list of the constructor. The code ssociated with these settings may often be found in the constructor code itself (within the braces {}) or in a separate method that is called form the constructor.

3. Below this are the model-implementation data. These data are typically specific populations of generic models. For example, the generic “Planet” model should be populated with Earth-specific data when the intent is to represent Earth. There are two schools of thought here:

    1. Create an EarthPlanet class that inherits from Planet and has its own model-level default-data. Instantiate an EarthPlanet

    2. Create an EarthDefaultData class whose sole purpose is to populate the generic Planet class. Instantiate a Planet class and a EarthDefaultData class and execute a default-data-class job to pulate the generic Planet instance with the specific data.

All JEOD models have one or more of these flavors of default-data, and they should not be edited.

Non-JEOD models may also use one or more of these design structures. Note that changes to default data (of any flavor) require that the simulation be recompiled to integrate the new data. The advantage of default-data over input-data is processing speed; if the same data is going to be used, unaltered, from simulation to simulation, it is faster to assign it to Default Data where it will be built it into the simulation at compile time rather than being read and populated through a Python interface every time a simulation starts.

Default Data files are typically written in C++ or C, with accompanying headers and source code. There are a limited number of examples in JEOD, used for defining values such as gravitational parameters. To include in an S_define, the header must be included with a *##include* statement, an instance of that class must be declared, and the defining method (we typically use *initialize*) called as a default data class job.

For example, to use the *GGM05C* gravity model, we would include the *earth_GGM05C* class.

models/environment/gravity/verif/SIM_csr_compare/S_define

```
...
 20 ##include "environment/gravity/data/include/earth_GGM05C.hh"
...
 25   jeod::SphericalHarmonicsGravitySource_earth_GGM05C_default_data grav_default_data;...
```

Then the *initialize* function must be written:

models/environment/gravity/data/src/earth_GGM05C.cc

```
...
32 void
33 SphericalHarmonicsGravitySource_earth_GGM02C_default_data::initialize (
34    SphericalHarmonicsGravitySource * SphericalHarmonicsGravitySource_ptr)
...
53    SphericalHarmonicsGravitySource_ptr->Cnm[  2] =
54       JEOD_ALLOC_PRIM_ARRAY (3, double);
55    SphericalHarmonicsGravitySource_ptr->Cnm[  2][  0] = -4.8416938905481E-04;
56    SphericalHarmonicsGravitySource_ptr->Cnm[  2][  1] = -2.0458338184745E-10;
...
```

Note that the *initialize* method takes an argument, a pointer to the S_define-defined instance of a *SphericalHarmonicsGravitySource*. This provides the *initialize* method with the information where to put the data it contains.

Modified Data

Modified Data is read at the start of every simulation. Large quantities of Modified Data can slow the launch of a simulation. However, if the data is likely to be altered between simulations, it may be faster to read in the specific value than to recompile the simulation each time (as would be necessary if it was Default Data). Data values stored in Default Data can be overwritten by also adding them to Modified Data. Modified Data are written in Python; they can take the form of a function definition, or simply variable assignments. Remember that if a function is being defined, it must be terminated with a return statement.

For example, in the *SIM_csr_compare* simulation we were just considering, it is necessary to define the state of the ISS. Since this is a set value for any given configuration, but varies with scenario, it is defined in Modified Data.

```
models/environment/gravity/verif/SIM_csr_compare/Modified_data/state/iss_typical.py
…
 9 def set_state_iss_typical(veh_obj_reference) :
10
11   #//
12   #// Set the translational position.
13   #//
14   veh_obj_reference.trans_init.dyn_subject = veh_obj_reference.body
15   veh_obj_reference.trans_init.reference_ref_frame_name = "Earth.inertial"
…
32   return
```

Input File Data

Each simulation run will have its own input file. Ultimately, it is the input file that is used to make the final assignment to all variables. The input file will pull in the Modified Data files, after which variables contained within the Modified Data files can be further modified. Input files are written in Python.

While it is possible to just enter all of the data into the input file, it is bad practice to do so. Large input files are difficult to maintain; a particular case-in-point is the situation in which it is desirable to propagate the same change to all runs of a given simulation. It is much easier to propagate changes to all runs when all input files pull in the same Modified Data file; changes are made by changing the one file rather than all of the input files.

If the Modified Data file defines a function, the file must first be included (with an *execfile* statement), then the function executed. There are two advantages to this method:

1. All of the includes can be put together at the top of the input file, then the functions executed as desired, which makes for a more tidy file

2. The function can receive an argument.

In our ongoing example, this method is used (although the *execfile* statement is kept with the function call) and the vehicle identification is passed to the function. That means that more than one vehicle could be populated with the same data by calling the same function with a different argument.

```
models/environment/gravity/verif/SIM_csr_compare/SET_test/RUN_01/input.py
...
 35 execfile( "Modified_data/state/iss_typical.py")
 36 set_state_iss_typical(sv_dyn)
...
```

If the Modified Data files are just data declarations, the input file need only source the Modified Data file (with the *execfile* command) to populate the variables. The advantage to this method is a smaller input file (1 line per Modified Data file instead of 2), but the file must be included at the correct location. The major disadvantage of this method comes when the same data can be applied to multiple instances; as a function call, each instance can be passed in to the same modified-data function in its argument list; with a simple data-file assignment, the data file must be replicated for each instance.

Repeating the same example, the same result would be obtained with this setup, with *sv_dyn* used explicitly in the Modified Data file rather than passed into it as an argument:

models/environment/gravity/verif/SIM_csr_compare/SET_test/RUN_01/input.py (edited)

```
...
 35 execfile( "Modified_data/state/iss_typical.py")
...
```

models/environment/gravity/verif/SIM_csr_compare/Modified_data/state/iss_typical.py (edited)

```
...
 9 # This line is not needed  “def set_state_iss_typical(veh_obj_reference) :”
10
11   #//
12   #// Set the translational position.
13   #//
14   sv_dyn.trans_init.dyn_subject = veh_obj_reference.body
15   sv_dyn.trans_init.reference_ref_frame_name = "Earth.inertial"
...
32 # no return needed without a function definition
```

Instantiating data in input

The simplest way to instantiate an instance of a class from the input-level (input file or Modified-data file) is to use the form:

```
instance = trick.ClassName()
```

(This will create a new instance of *ClassName*, called *instance).*

Note that there are some limitations to this capability.

1. The header file for the class must have been included (##include) in the S_define
2. If the class contains data that requires units, then in many – but not all – cases either an instance, or an instance-pointer must have been declared in the S_define. Without that step, Trick will not be able to interpret the units on the data used to populate the new instance. Note that it it NOT necessary to use the S_define-declared instance, it just has to be there.
3. In many cases, it is required that such an instantiation is followed with

   ```
   instance.thisown = 0
   ```

   This preserves the instance when it passes out of scope – such as when the instantiation is performed in a function which then returns to the main input file, or when the instance is loaded onto an array of elements for later use in JEOD. Even when this is not required, it does no harm to include it. If in any doubt at all, include it.

Scheduling Events

Frequently it is necessary to schedule certain function calls based on a certain time or event. To schedule based on time, an activity we will use quite a bit in the course exercises, use the *read* command. There is potential for some confusion here for readers looking at the JEOD verification simulations for recommended practice (by the

way, doing so is generally not recommended; the verification simulations are not always good examples to use as building blocks). There are two forms to the read statement:

```
trick_ip.ip.add_read(...)
trick.add_read(...)
```

These two forms do the same thing.

The form mostly used in JEOD for read statements is as follows:

```
trick.read(time, """
command
""")
```

Note that the command goes on its own line, there are three double quotes before and after the command, and there can be no spaces before the command on the command line. The triple-quotes are a Python syntax that allow multiple lines of code to be interpreted as a single command block.

A.3 Output Data

The input file must also specify which variables to log as the simulation progresses. Typically, as with Modified Data, that is stored in a separate file in a separate directory and included in the input file. These files are also written in Python.

Conventionally, these *Log Data* files define a function in much the same way that the Modified Data files did, and are called from the input file with an argument that specifies the frequency with which the data in that file should be logged.

```
models/environment/gravity/verif/SIM_csr_compare/SET_test/RUN_01/input.py
...
15 LOG_CYCLE = 1.0
...
18 execfile( "Log_data/log_sv_state_rec.py")
19 log_sv_state_rec( LOG_CYCLE )
...
```

The Log Data file starts by declaring the name of the function. Then the following values are set:

1. *recording_group_name*, set to any unique name.
2. *dr_group*, set to an automatic expression of the recording_group_name.
3. *dr_group.thisown = 0* keeps the group active after the function returns
4. *dr_group.set_cycle(log_cycle)* sets the logging period to the input argument from the input file
5. *dr_group.freq = trick.sim_services.DR_Always*

Then the desired variables are added one at a time with the *dr_group.add_variable* call. For arrays, the values can be entered in a for loop, as illustrated below. Note that in Python, indentation is VERY important; notice that entry on line 23 is indented, indicating that it is a part of the for-loop started at line 22. To exit the for-loop, return to the earlier indentation.

Finally, the group is added to the collection of log data, and a return is necessary because this defines a function.

models/environment/gravity/verif/SIM_csr_compare/Log_data/log_sv_state_rec.py

```
...
16 def log_sv_state_rec ( log_cycle ) :
17   recording_group_name =  "State"
18   dr_group = trick.sim_services.DRBinary(recording_group_name)
19   dr_group.thisown = 0
20   dr_group.set_cycle(log_cycle)
21   dr_group.freq = trick.sim_services.DR_Always
22   for ii in range(0,3) :
23     dr_group.add_variable("sv_dyn.body.composite_body.state.trans.position[" + str(ii) +
    "]" )
...
27   data_record.drd.add_group(dr_group)
28
29   return
```

A.4 Data Analysis

There are several tools available for analyzing data. For the purposes of this course, trick_qp is sufficient but not required. Exercise solutions are written with examples provided from trick_qp.

Appendix B Computing Overview

Useful commands:

- *ls* lists all files in current directory
- *cd* changes directory
  - *cd* .. changes directories up one (to its parent)
  - *cd* <*directory*> changes to a sub-directory / folder
  - *cd* <*dir1*>/<*dir2*> changes down 2 levels
- *[TAB]* auto-completes commands and filenames
- *trick-CP* compiles a simulation

All JEOD source, data, etc. files are text files. Edit with any basic text editor, e.g.

- vi
- emacs
- Kate
- KWrite
- KEdit

Some editors are opened as a separate window, some are opened from the command line, many can be used in either mode. JEOD does not recommend a particular text editor

Appendix C Introduction to C++

C.1 Classes

Contents

- **Data Elements** - the collection of data elements is much like a C struct in appearance. Elements may be primitive (e.g. int, double), or classes
- **Methods** - a.k.a. functions. These are the processes by which the data in the class is used and re-populated.

Header Layout

Headers in JEOD typically found in *include/<class_name>.hh.* Typically (though not universally), headers are organized by:

1. Data Elements
    1. public
    2. protected
    3. private
2. Methods
    1. public
    2. protected
    3. private

Accessibility (Public - Protected - Private)

By accessible, we mean whether:

- Data can be accessed as readable, writable, both, or neither.
- Methods can be called.

C++ has limited distinction between readable and writeable data, but utilizes the flag *const* to make readable data non-writeable. Apart from that, C++ provides three levels of protection that determine whether elements are accessible.

Public

Variables and methods are accessible from anywhere

Protected

Variables and methods are accessible only from the class in which they are defined, or from any subclass of that class (we will look at the concept of inheritance and subclasses a little later).

Private

Variables and methods are accessible only from the class in which they are defined

Friend

A declaration made in class A that class B is a Friend means that class B has access to everything in class A

Is-a and Has-a

This distinction becomes important in the next C++ concept

At a primitive level,

```
class DummyClass {
        double dummy_variable;
}
```

dummy_variable **is a** *double*.

DummyClass **has a** *double* (called *dummy_variable*).

At a derived level:

```
class DummyClass_v2 {
   DummyClass dummy_class_instance; }
```

dummy_class_instance **is a** *DummyClass*. it **has a** *double* (called *dummy_variable).*

DummyClass_v2 **has a** *DummyClass* (called *dummy_class_instance*)

C.2 Subclasses and Inheritance

A **Base-Class** defines all elements, one at a time, elements may be primitives or instances of declared classes.

A **Sub-Class** inherits from another class (e.g. a base-class). By inheriting from its parent, it "has" almost everything that its parent has, and can add more. The only things it does not receive by inheritance are those marked "Private" in the parent class. Private data in the parent can only be accessed through (public or protected) methods inherited from parent, private methods in the parent class cannot be accessed. Additional data elements and methods can be defined, and methods may be replaced under certain situations (see Polymorphism below).

In general, the sub-class instance **is a** type of the parent-class, but the converse is not true.

Example

Define class DummyA to have a single data element (*dummy_a_var*) and a single method (*test*) that takes no argument and returns no result.

```
class DummyA {
public:
double dummy_a_var;
void test(void);
}
```

Define class DummyB1 to inherit from class DummyA, and to add a single data element (*dummy_b_var*)

```
class DummyB1 : public DummyA {
public:
double dummy_b_var;
}
```

Define class DummyB2 to contain an instance of DummyA, *dummy_a_instance*, and the data element, *dummy_b_var.*

```
class DummyB2 {
public:
DummyA dummy_a_instance;
double dummy_b_var;
}
```

In this example, *DummyB1* and *DummyB2* both have the same data and methods, but they are accessed differently.

DummyB1 gets *dummy_a_var* and *test* by inheritance, so contains the elements:

- *dummy_a_var*
- *dummy_b_var*
- *test()*

DummyB2 gets *dummy_a_var* and *test* by inclusion, so contains the elements:

- *dummy_a_instance.dummy_a_var*
- *dummy_b_var*
- *dummy_a_instance.test()*

Scope of Methods

Scope refers to the extent to which methods can access data, methods can utilize any data within their scope. Scope can be specified, or it defaults to “this” (the class in which the code is operating at the time the method call is made).

Example

Class *Vehicle* has an integer value, *count*, and a method, *update*.

```
class Vehicle {
public:
  int count;
  void update(void);
}
```

The method *update* is defined such that it increments *count*:

```
void
Vehicle::update(void) {
  count++;
  return;
}
```

Whenever the method *Vehicle::update()* is called, it will increment the variable *count* that is in the class associated with the particular instance of *Vehicle*.

Now define a new class, *VehicleManager:*

```
class VehicleManager {
public:
  Vehicle veh;
  void update(void);
}
```

and define this update method:

```
void
VehicleManager::update(void) {
  veh.update();
  return;
}
```

Note that the two *update* methods now defined have different scopes (one is *Vehicle* and the other *VehicleManager)*, so even though they have the same name, they are distinguishable. Indeed, methods are distinguished by three values – name, scope, and argument list. Only if all three are identical will the code fail to recognize them as different entities.

Now, running *VehicleManager::update()* will call *Vehicle::update()*, which will increment the variable *count.* The effect in *VehicleManager* is that *veh.count* will increment.

Inheriting methods

Continuing with the current example, define a new manager class that inherits from the old:

```
class VehManagerNew : public VehicleManager {
public:
  Vehicle new_veh;
}
```

This class now has two vehicles – *veh* (by inheritance) and *new_veh* (by inclusion) – and the method *update()* (by inheritance).

Running the method *VehManagerNew::update()* will still increment *veh.count*. It will not affect *new_veh.count* in any way, because the method it inherited refers specifically to *veh.count*. To change *new_veh.count*, we would have to rewrite the method, and for that we need to consider virtual methods and polymorphism.

Scope of Locally Declared Variables

When declaring variables locally in a C++ method, the scope of those methods is limited to the current method. Once the method exits/returns, those variables fall out of scope and cease to exist.

Overloading Methods

A method is identified by name, scope, and argument list. Creating multiple instances of a function with the same name and same scope (but different argument list) is referred to as overloading the method.

C.3 Virtual Methods and Polymorphism

A method is made **virtual** with the addition of the prefix keyword *virtual*. A virtual method can be redefined in sub-classes. It leads to the concept of **Polymorphism** (multiple variants), in which the operation of a method depends on the scope on which it is called.

Continuing with the current example, suppose the class *VehicleManager* had been defined with a virtual method instead:

```
class VehicleManager {
public:
  Vehicle veh;
  virtual void update(void);
}
```

Now, when we define the subclass VehManagerNew, we can either simply inherit the base-class method (default, no action needed, the “virtual” in this case has no effect), or redefine it. To redefine it, it must be re-declared in the header file:

```
class VehManagerNew : public VehicleManager {
public:
  Vehicle new_veh;
  virtual void update(void);
}
```

Note that it is considered good practice, though not required, to propagate the keyword *virtual* to the subclass method declaration.

Suppose that we want to make this new method increment both *count*s – that of *veh* and *new_veh.* That can be accomplished in two different ways:

1. Go back to the beginning. Since this class contains both *veh* and *new_veh*, the respective *update* methods can be called directly. Notice that prefix *veh* (or *new_veh*) scopes the *update* call appropriately.

```
void
VehManagerNew::update(void) {
  veh.update();
  new_veh.update();
  return;
}
```

2. Build on the existing sub-class function. The base-class function already updates *veh*, so it is sufficient to call that, then add the new functionality to update *new_veh*. Notice that the first call is scoped to the base-class.

```
void
VehManagerNew::update(void) {
  VehicleManager::update();
  new_veh.update();
  return;
}
```

Abstract Classes

Sometimes, want to define common functions and data for multiple similar classes, but the common aspects alone may be insufficient to ever justify creating an instance with those alone. In this case, we put the aspects together into a non-instantiable, or **abstract** class.

An abstract class is defined as such by having at least one pure-virtual method (a method that by its definition is null).

```
virtual void dummy_method(void) = 0;
```

When this abstract class is inherited, the pure-virtual method may be redefined; only when all pure-virtual methods have been redefined can a class be instantiated. A class that only redefines a subs et of its parent pure-virtual methods is still abstract.

Example

```
class AbstractA {
   virtual void dummyA(void) = 0;
   virtual void dummyB(void) = 0;
}
```

AbstractA has two pure-virtual methods, so is abstract.

```
class AbstractB : public AbstractA {
   virtual void dummyA(void);
}
```

AbstractB inherits both pure virtual methods, but redefines one of them. It still has one, so is abstract.

```
class Instantiable : public AbstractB {
   virtual void dummyB(void);
}
```

Instantiable inherits one pure virtual method and redefines it. It is instantiable (no longer abstract)

Appendix D Frequently Used Math Operations

This appendix provides 'cheat-sheets' for the mathematical operations provided by, and used within, JEOD. There are four classes with mathematical functions illustrated here – Numerical, Vector3, Matrix3x3, and Quaternion. The last 5 pages of this document are formatted for easy printing and subseqeunt reference as a bi-fold sheet.

D.1 How to Access Methods

For the first three classes (Numerical, Vector3, Matrix3x3), methods are typically called directly with values either assigned to elements of the argument list, or returned.
e.g.

```
double x_sq = Numerical::square(x);

double mx_transpose[3][3];
Matrix3x3::transpose(mx, mx_transpose)
```

The methods for the fourth class, Quaternion, are called from within an instance of the JEOD-provided class Quaternion.

e.g.

```
Quaternion quat_1, quat_2;
quat_1.make_identity();
quat_1.conjugate(quat_2);
```

For all four classes, the appropriate header files must be included.

D.2 Nomenclature

A letter followed by a number is an input, a letter alone is an output.

S – scalar	all scalars are double
V – vector	all vectors are double[3]
W – 4-vector	all such arrays are double[4] (used for quaternions only)
M – matrix	all matrices are double[3][3]
Q – Quaternion	all quaternions comprise a double *scalar* and a double[3] *vector.*

In addition, for quaternion methods only, q, s, and v represent this calling quaternion and its scalar and vector components respectively.

Numerical::

(internal)

fabs (S1)
returns a double equal to absolute value of S1

square(S1)
returns a double = $S1^2$, with underflow protection

square_incr(S1, S2)
returns S2 = S2 + $S1^2$, with underflow protection on $S1^2$. S2 is overwritten.

Vector3::

(initialize)

initialize(V)
produces a zero-vector.

fill (S1, V)
produces a vector with values = S1

unit (V, S) (S<=3)
V = unit vector with a 1 at position S1

copy (V1, V)
V = V1

(internal)

vmagsq (V1)
returns a double = square of magnitude of V1

vmag(V1)
returns a double equal to magnitude of V1

normalize (V1)
returns a normalization of V1, or 0 if V1 has 0 magnitude.

normalize(V1, V)
V = normalized V1, or 0 if V1 has 0 magnitude.

zero_small (S, V1)
zeros any values of V1 that are $|V1_i| < S$

(addition)

sum(V1, V2, V)
V = V1 + V2

sum(V1, V2, V3, V)
V = V1 + V2 + V3

diff (V1, V2, V)
V = V1 - V2

decr(V1, V2)
V2 = V2 – V1

decr(V1, V2, V3)
V3 = V3 - (V1 + V2)

incr(V1, V2)
V2 = V1 + V2

incr(V1, V2, V3)
V3 = V1 + V2 + V3

Vector3::

(multiplication)

scale (S1, V1)
V1 = V1 * S1

scale (V1, S1, V)
V = V1 * S1

negate (V1)
V1 = -V1

negate (V1, V)
V = -V1

dot(V1, V2)
returns a double equal to dot-product of V1 and V2

cross(V1, V2, V)
V = V1 x V2

(compound)

scale_incr(V1, S1, V2)
V2 = V2 + V1*S1

scale_decr(V1, S1, V2)
V2 = V2 – V1*S1

cross_incr(V1, V2, V3)
V3 = V3 + V1 x V2

cross_decr(V1, V2, V3)
V3 = V3 - V1 x V2

(matrix operations)

transform (M1, V1, V)
V = M1 * V1

transform (M1, V1)
V1 = M1 * V1

transform_transpose (M1, V1, V)
V = $M1^T$ * V1

transform_transpose (M1, V1)
V1 = $M1^T$ * V1

transform_incr (M1, V1, V2)
V2 = V2 + (M1 * V1)

transform_transpose_incr (M1, V1, V2)
V2 = V2 + ($M1^T$ * V1)

transform_decr (M1, V1, V2)
V2 = V2 - (M1 * V1)

transform_transpose_decr (M1, V1, V2)
V2 = V2 - ($M1^T$ * V1)

Matrix3x3::

(initialize)

initialize (M)
creates M = zero matrix

identity (M)
creates M = identity matrix

copy (M1, M)
M = M1

print (M1)
prints M1 to stderr

(internal)

transpose (M1)
$M1 = M1^T$

transpose (M1, M)
$M = M1^T$

invert (M1, M)
$M = M1^{-1}$

invert_symmetric (M1, M)
$M = M1^{-1}$ when M1 is symmetric

(addition)

add (M1, M2, M)
M = M1 + M2

subtract (M1, M2, M)
M = M1 - M2

incr (M1, M2)
M2 + M2 + M1

decr (M1, M2)
M2 = M2 - M1

(multiplication)

negate (M1)
M1 = - M1

scale (S1,M1)
M1 = M1 * S1

product (M1, M2, M)
M = M1 * M2

product_left_transpose (M1, M2, M)
$M = M1^T * M2$

negate (M1, M)
M = -M1

scale (M1, S1, M)
M = M1 * S1

product_right_transpose (M1, M2, M)
$M = M1 * M2^T$

product_transpose_transpose (M1, M2, M)
$M = M1^T * M2^T$

(vector operations)

cross_matrix (V1, M)
M = skew-symmetric cross product matrix
such that (M * v) = (V1 x v)

outer_product (V1, V2, M)
M = V1 (column) * V2 (row)

(matrix operations)

transform_matrix (M1, M2, M)
$M = M1 * M * M1^T$

transpose_transform_matrix (M1, M2, M)
$M = M1^T * M * M1$

Quaternion::

(construction)

()
sets s = 1, $\vec{v}$ = 0.

(S1)
sets s = S1, $\vec{v}$ = 0.

(M1)
sets q equal to transformation matrix M1

(initialize)

set_to_zero ()
sets s = 0, $\vec{v}$ = 0.

make_identity ()
sets s = 1, $\vec{v}$ = 0.

copy_to(W)
copies quaternion to W.

copy_from(W1)
copies values from W, s = W[0], $\vec{v}$ = W[1-3]

normalize()
makes quaternion unit-quaternion, with $s \geqslant 0$

normalize_integ()
makes quaternion unit-quaternion, with no sign restriction on s.

normalize(Q)
Q = unit-quaternion of q, with $S \geqslant 0$

normalize_integ(Q)
Q = unit-quaternion of q, with no sign restriction on S.

(internal)

norm_sq()
returns $s^2 + \vec{v} \cdot \vec{v}$

conjugate()
$q = q^*$

conjugate(Q1)
$Q1 = q^*$

Quaternion::

(multiplication)

scale(S1)
$q = q \times S1$

scale(S1, Q)
$Q = q \times S1$

multiply (Q1, Q)
$Q = q \times Q1$

multiply(Q1)
$q = q \times Q1$

conjugate_multiply (Q1, Q)
$Q = q^{*} \times Q1$

conjugate_multiply (Q1)
$q = q^{*} \times Q1$

multiply_conjugate (Q1, Q)
$Q = q \times Q1^{*}$

multiply_conjugate(Q1)
$q = q \times Q1^{*}$

multiply_left(Q1)
$q = Q1 \times q$

multiply_left_conjugate(Q1)
$q = Q1^{*} \times q$

multiply_vector_left (V1, Q)
$Q = [0, V1] \times q$

multiply_vector_right (V1, Q)
$Q = q \times [0, V1]$

(rotation)

left_quat_from_eigen_rotation(S1, V1)
makes quaternion represent input eigen-rotation of S1 radians about V1.

left_quat_to_eigen_rotation (S, V)
produces eigen-angle S, eigen-vector V decomposition of q.

left_quat_from_transformation(M1)
makes quaternion represent transformation matrix.

left_quat_to_transformation(M1)
makes transformation matrix from quaternion q.

eigen_compare (Q1, S, V)
produces eigen-angle S, eigen-vector V decomposition of $q \times Q1^{*}$