EMTGv9 Constraint Scripting

Donald H. Ellison [*], Jacob A. Englander [†], Ryo Nakamura [‡], Noble Hatten [§]

Revision Date	Author	Description of Change
November 20, 2018	Donald Ellison	Listing of current implemented constraints
January 16, 2019	Jacob Englander	Updated to reflect new organization of .emtgopt file. Added overview section.
December 21, 2018	Jacob Englander	Added phase distance constraint, boundary distance constraint, MGAnDSMs maneuver constraints, PSFB thrust angle constraint, and PSFB duty cycle control
January 16, 2018	Jacob Englander	Added introductory material and instructions for how to use the scripted constraints with the new modular options file format.
January 23, 2018	Jacob Englander	Added RBP, RPB, RRP, and RPR angle constraint documentation.
February 5, 2019	Jacob Englander	Updated to new options file spec, commentable constraints, and BPT angle with constant bounds.
February 11, 2019	Jacob Englander	Added BCF latitude and longitude boundary constraints. Changed parallel shooting constraint text to recognize that PSFB is no longer the only parallel shooting phase type.
February 14, 2019	Jacob Englander	Added velocity declination constraint.
February 28, 2019	Jacob Englander	Added vertical flight path angle and velocity magnitude constraints.
October 10, 2019	Ryo Nakamura	Added spinning body-relative velocity magnitude constraint.
October 23, 2019	Ryo Nakamura	Added latitude, detic altitude, spherical velocity azimuth, relative velocity azimuth, and relative velocity HFPA constraint.
February 12, 2021	Noble Hatten	Created "How-To" section for creating a new scripted constraint.
July 29, 2021	Noble Hatten	Added two-body rotating frame state boundary constraint.
July 30, 2021	Noble Hatten	Added new content to how-to section.
September 8, 2021	Noble Hatten	Added PyEMTG interface description.

[*]Aerospace Engineer, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

[†]Aerospace Engineer, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

[‡]Aerospace Engineer, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595, detailed from JAXA

[§]Aerospace Engineer, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

Contents

List of Acronyms

AU Astronomical Unit

JD Julian Date

MJD Modified Julian Date

NLP nonlinear program

MGALT multiple gravity assist with low-thrust

MGAnDSMs multiple gravity assist with n deep-space maneuvers using shooting

PSFB Parallel Shooting with Finite-Burn

PSBI Parallel Shooting with Bounded Impulses

FBLT finite-burn low-thrust

BCF body-centered fixed

BPT body-probe-thrust

4PL four parameter logistic

BCF body-centered fixed

1 Overview

EMTGv9 allows the user to specify a wide range of scripted constraints beyond what is available in the GUI. This document describes the three classes of constraints - boundary constraints, maneuver constraints, and phase distance constraints (also known as path constraints).

EMTGv9's .emtgopt input file contains a separate, copyable block for each journey in the mission. At the end of each journey, blocks exist for each of the three classes of constraints as shown below:

```
#Maneuver constraint code
#Works for absolute and relative epochs and also magnitudes
BEGIN_MANEUVER_CONSTRAINT_BLOCK
END_MANEUVER_CONSTRAINT_BLOCK


#Boundary constraint code
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
END_BOUNDARY_CONSTRAINT_BLOCK


#Phase distance constraint code
BEGIN_PHASE_DISTANCE_CONSTRAINT_BLOCK
END_PHASE_DISTANCE_CONSTRAINT_BLOCK
```

Constraints are always prefixed with a tag representing the phase that they constrain. Tags are indexed from zero. For example, constraints on the first phase of the journey are prefixed `p0`, and constraints on the fifth phase are prefixed `p4`. The user can also place constraints on the *last* phase of the journey using the prefix `pEnd`.

Internally, the constraints are stored in the JourneyOptions structure as lists of strings:

```
JourneyOptions.ManeuverConstraintDefinitions
JourneyOptions.BoundaryConstraintDefinitions
JourneyOptions.PhaseDistanceConstraintDefinitions
```

PyEMTG collects the journey-level lists of constraints into "master" constraint lists that exist at the mission level as shown below. To do this, call `MissionOptions.AssembleMasterConstraintVectors`. You can also disassemble the master constraint list back to the journey level via

`MissionOptions.DisassembleMasterDecisionVector`.

```
MissionOptions.ManeuverConstraintDefinitions
MissionOptions.BoundaryConstraintDefinitions
MissionOptions.PhaseDistanceConstraintDefinitions
```

If you wish to edit scripted constraints at the journey level via Python, do the following:

```
MissionOptions.DisassembleMasterDecisionVector()
<your constraint edits here>
MissionOptions.AssembleMasterConstraintVectors()
```

Alternatively, if you wish to edit at the mission level, perform the procedure below. However, you will need to prefix `jJ` to the constraint names when manipulating the master constraint definition lists, where `J` is the index of the journey that you wish to edit.

```
MissionOptions.AssembleMasterConstraintVectors()
<your constraint edits here>
MissionOptions.DisassembleMasterDecisionVector()
```

Remember to call `MissionOptions.DisassembleMasterDecisionVector` and `MissionOptions.AssembleMasterDecisionVector` as appropriate so that the mission and journey-level constraint definition lists remain in sync. In a future update, we will find a way to make this happen automatically.

The user may comment out a constraint with the `#` character. The constraint will remain in the script through ingest and rewrite in PyEMTG and EMTG, but will not be used in optimization.

2 Boundary Constraints

Boundary constraints may be applied to any boundary condition class. In order to apply the constraint to the left boundary of a particular phase/journey, the constraint should be tagged as a departure constraint (e.g. p2_departure_orbitperiod_1yr_1.1yr). Likewise, for a right boundary, it would be tagged as "arrival".

Moving forward, the ability to apply the constrain infinitesimally before or after the boundary of choice is also being added by appending a "-" or "+" to the boundary name, yielding four possibilities: "departure-", "departure+", "arrival-", and "arrival+". The difference between "-" and "+" is that the "-" version does not take into account events that happen instantaneously at the boundary (like a Δv or a mass drop), while the "+" version does. So far, the state in two-body rotating frame constraint is the only constraint that supports this capability and syntax. For constraints that do not support this syntax, the user must look at the `process_constraint` method of the boundary constraint to determine whether the constraint is evaluate before or after the boundary event.

Boundary constraints are specified in the boundary constraint block at the end of each journey in the .emtgopt file. (See Section 1.)

2.1 How to Create a New Boundary Constraint

Creating a new boundary constraint consists of creating a new class derived from `SpecializedBoundaryConstraint` and editing existing code to make EMTG aware of the existence of the new constraint. All required steps are described in this section.

The easiest way to create a new specialized boundary constraint is to examine (i.e., copy and paste) an existing boundary constraint to use as a starting point. As of July 30, 2021, it is recommended to start from the `BoundaryStateInTwoBodyRotatingFrameConstraint` class because it contains most if not all of the features required to create a new boundary constraint.

2.1.1 New Files

A new boundary constraint requires creating a new class derived from `SpecializedBoundaryConstraintBase`. Thus, a `.h` and a `.cpp` file must be created. These files are placed in

`emtg/src/Mission/Journey/Phase/BoundaryEvents/SpecializedBoundaryConstraints/`

In addition, the CMakeLists.txt file in the preceding directory must be edited so that the new header file is added to the list of `SPECIALIZED_BOUNDARY_EVENT_HEADERS` and the new source file is added to the list of `SPECIALIZED_BOUNDARY_CONSTRAINTS_SOURCE`. The EMTG cmake file must be reconfigured and regenerated after the new files are added to the CMakeLists.txt file in order for the changes to take effect.

2.1.2 Structure of the New Class

Several methods of `SpecializedBoundaryConstraintBase` must be overridden by the new constraint class:

- `output`: Prints the name and value of the constraint to the .emtg output file. The text is displayed underneath the state and control history output for the journey on which the constraint is applied.
- `calcbounds`: Parses the text of the constraint definition; sets the upper and lower bounds of the constraint; sets the sparsity pattern of the constraint. Called once per execution, prior to optimization.
- `process_constraint`: Calculates the value of the constraint and, optionally, its Jacobian.

In addition, it may be useful to add code to the constraint's constructor method. E.g., the user may wish to set a reference frame.

It is common practice to create class variables for a boundary constraint class. In particular, a class variable for the value being constrained can be used in the `output` method to print the value.

2.1.3 The Constructor Method

The constructor method is constraint-specific. There are two common argument lists for constructors: one that contains the `constraint_reference_frame` argument and one that does not. The `constraint_reference_frame` argument is required if the user is allowed to specify the reference frame in which the constraint is applied. An example is `InclinationConstraint`, which may be found in the `OrbitElementsConstraints` directory. The `constraint_reference_frame` argument is not required if the user is not allowed to specify the reference frame in which the constraint is applied. An example is `SemimajorAxisConstraint`, which may be found in the `OrbitElementsConstraints` directory.

In some cases, the contents of the constructor method is empty. Examples of tasks that are commonly performed in a constructor include:

- Assigning the `constraint_reference_frame` argument to a class variable, if relevant.
- Setting the dimensions of any `Matrix` class variables.
- Switching on the computation of orbit elements at the boundary event. See Section 2.1.9 for more details.

2.1.4 The `output` Method

The `output` method is extremely simple and consists of printing the value of the constraint and, if desirable, basic associated metadata (e.g., reference frame in which the constraint was computed), to the .emtg output file. The text is displayed underneath the state and control history output for the journey on which the constraint is applied.

2.1.5 The `calcbounds` Method

The first task of `calcbounds` is to parse the text of `this->constraintDefinition`, which contains the name of the constraint, when the constraint should be applied, the upper and lower bounds of the constraint, and any additional information required to specify the constraint. The different parts of the constraint definition are separated by underscores, and the `boost::split` function is used to extract the individual parts.

The text description of the constraint is set with

```
this->Fdescriptions->push_back(prefix + "<description>")
```

where `<description>` is a short description of the constraint. In the `SpecializedBoundaryConstraintBase` `setup_calcbounds` method, `prefix` is set to `this->name + ":  "`. Note: Do not put a comma in any of the text that will be part of `Fdescriptions` because it will mess up csv readers.

The lower bound of the constraint is set with

`this->Flowerbounds->push_back(<lowerBound>)`

The upper bound is set with

`this->Fupperbounds->push_back(<upperBound>)`

Important: If it is desired to scale the constraint, then the values added to `Flowerbounds` and `Fupperbounds` must be scaled appropriately.

The final job of `calcbounds` is to set the sparsity pattern of the constraint function. This process can be quite confusing, so copy/pasting and making minimal adaptations for a specific constraint from a previous boundary constraint class is highly recommended. This walkthrough assumes that the reader is following along with the code in `BoundaryStateInTwoBodyRotatingFrameConstraint.cpp`.

The first call when creating the sparsity pattern is

```
std::vector< std::tuple<size_t, size_t, double> >&
 Derivatives_of_StateAroundEvent = this->myBoundaryEvent->
get_Derivatives_of_StateBeforeOrAfterEvent(this->getStateAndDerivativesIndex);
```

The first element of each returned tuple in `Derivatives_of_StateAroundEvent` is the index of the decision variable in the global decision vector. The second element of each returned tuple in `Derivatives_of_StateAroundEvent` is the local position of the decision variable in the state vector. Note that the position/velocity of the state vector is always in ICRF Cartesian coordinates, regardless of the frame and state representation used to encode the decision variables. The third element of each returned tuple in `Derivatives_of_StateAroundEvent` is the actual derivative value. When `calcbounds` is called, no value has been set, but that is OK —we do not need it for setting the sparsity pattern. We can think of the elements of `Derivatives_of_StateAroundEvent` as: the derivative of the second element with respect to the first element is equal to the third element.

We do this for derivatives with respect to time, as well:

```
std::vector< std::tuple<size_t, size_t, double> >&
Derivatives_of_StateAroundEvent_wrt_Timethis->myBoundaryEvent->
get_Derivatives_of_StateBeforeOrAfterEvent_wrt_Time(this->getStateAndDerivativesIndex);
```

We then loop through state indexes. In the two-body rotating frame constraint example, this is done by looping from 0 to 2, and handling velocity separately from position within the same loop. An alternative formulation could loop from 0 to 5.

Within the loop, the first action handles the derivatives with respect to position. We loop through each element of the `Derivatives_of_StateAroundEvent` vector. For each element, we check if second element of that tuple is equal to our current state index (the loop variable of our

outer loop). In other words, does our current index correspond to differentiating an element of the position state? If so, we wish to add a sparsity entry for it because our constraint depends on all elements of the position vector. So, we call

```
this->create_sparsity_entry(this->Fdescriptions->size() - 1,
std::get<0>(Derivatives_of_StateAroundEvent[dIndex]),
state_Gindex_constraint_position_wrt_StateAroundEvent_variables);
```

The first argument to this call is the index of the current constraint in the overall vector of constraints. The second argument is the position of the independent variable currently under consideration in the overall vector of decision variables. The third argument is an output: the vector of global Jacobian entry indexex, which is a single-dimension vector because it is an unrolled vector.

After looping through the elements of the `Derivatives_of_StateAroundEvent` vector, we call

```
this->dIndex_constraint_position_wrt_StateAroundEvent.push_back
(state_dIndex_constraint_position_wrt_StateAroundEvent);

this->Gindex_constraint_position_wrt_StateAroundEvent_variables.push_back
(state_Gindex_constraint_position_wrt_StateAroundEvent_variables);
```

These calls basically put what were originally local variables into class variables that can be accessed from later calls to `process_constraint`.

We then do a similar loop to create sparsity entries for "implicit" dependencies of state elements on time variables. These exist because we are using a multiple-shooting method. The state and time loops are separate because we can have non-zero partial derivatives with respect to time variables from other phases in a multiple-shooting algorithm, whereas we will never have non-zero partial derivatives with respect to state variables from other phases. For example, the position state at a boundary depends on the launch epoch and the times of flight of all previous phases.

After handling the dependencies of position on state and time decision variables, two more loops are performed to do the same thing for velocity. In the two-body rotating frame example, the constraint only depends on velocity if a velocity state is being constrained. As a result, the velocity sparsity pattern creation loops are in an `if` block.

After completing the loop through state indexes, we still need to handle *explicit* time dependencies, if they exist. An explicit time dependence exists if the constraint value depends on an ephemeris lookup or a time-dependent rotation matrix. If explicit time dependencies exist, we call

```
std::vector<size_t> timeVariables = this->myBoundaryEvent->get_Xindices_EventRightEpoch();
```

and then loop through `timeVariables`, creating sparsity entries. Note that this workflow only works because EMTG constructs constraints sequentially in the order in which time marches forward. At the time `calcbounds` is called for a constraint, subsequent journeys haven't been constructed yet.

The `calcbounds` of each specialized constraint is only called once in each EMTG execution, prior to optimization.

2.1.6 The `process_constraint` Method

The `process_constraint` method is called whenever EMTG needs to evaluate the value of the constraint and/or its Jacobian.

The arguments are:

- `X`: The decision vector. Often, `X` is not used directly because more useful routines exist to do things like extract the state at the boundary.
- `Xindex`: This argument is not actually used, and, confusingly, is overridden locally in the `process_constraint` method of many constraints.
- `F`: Vector of constraints. The value of the constraint is added to this vector.
- `Findex`: A locator of a constraint within the vector of constraints `F`.
- `G`: The Jacobian of the constraint with respect to all decision variables. Must be populated if `needG` is true.
- `needG`: If true, the Jacobian of the constraint must be calculated. If false, it need not be.

The value of the constraint must be calculated. Recall that the value of the constraint is generally assigned to a class member so that it is accessible from the `output` method. Methods exist for doing things like obtaining the state of the spacecraft at the point of constraint application. Once the value is calculated, it is added to the vector of constraints with

```
F[Findex++] = <value>
```

If the Jacobian is required, it must be calculated. Frequently, the dependencies of a constraint are some subset of position, velocity, and time at, infinitesimally before, or infinitesimally after a boundary event.[1] A new specialized boundary constraint is responsible for calculating these "first-level" derivatives, and, then, using the chain rule to achieve the derivatives of the constraint with respect to the decision variables themselves. The derivatives of the state vector with respect to the decision variables at the boundary event are accessible to facilitate this procedure.

If the Jacobian is required, it must be placed in the correct entries of `G`. Determining the correct entries of `G` is similar to the procedure used to set the sparsity pattern in `calcbounds`. Like for setting the sparsity pattern, this walkthrough assumes that the reader is following along with the code in `BoundaryStateInTwoBodyRotatingFrameConstraint.cpp`. We follow these steps:

[1] "Infinitesimally before/after" means before/after an instantaneous boundary event, such as a Δv, takes place.

1. Prior to setting the relevant elements of `G`, it is standard practice to set them to zero. We loop through the relevant state indices and the relevant entries of `this->Gindex_constraint_position_wrt_State` to know which elements to set to zero. We also loop through the explicit time dependence indexes, if relevant.

2. We then set the derivatives of the constraint with respect to non-time variables affecting the boundary state. We grab the derivatives of the state at the boundary with respect to decision variables because this is the last link in the chain rule chain we need to create. Then, we loop through various indexes to place our derivatives appropriately. The things we need to keep in mind are:

   - Our `Gentry` needs to contain our local derivative entry (constraint with respect to boundary state) multiplied with the derivative of the state at the boundary with respect to the relevant decision variable, which is held in `std::get<2>(Derivatives_of_StateAroundEvent[dIndex])`
   - When we add our `Gentry` to `G[Gindex]`, the premultiplication by `this->X_scale_factors->operator[]` takes care of scaling the decision variable, but it is our responsibility to scale by the scale factor of the constraint, if one was used.

3. We then set derivatives of the constraint with respect to time variables affecting the boundary state (i.e., the implicit time dependencies). This procedure is extremely similar to that for setting the derivatives with respect to the boundary state itself. The important thing to remember here is that our local part of the chain rule still uses derivatives of our constraint with respect to the boundary state, *not* time. Time dependence comes through the other part of the chain rule: the time derivatives of the boundary state.

4. If necessary, we then set explicit time dependencies. This is where derivatives of the constraint directly with respect to time are used. There is no additional chain rule beyond that here.

2.1.7 Updating `SpecializedBoundaryConstraintFactory`

In addition to creating a new class derived from `SpecializedBoundaryConstraintBase`, a new block of code must be added to the `create_boundary_event_constraint` function in the file `SpecializedBoundaryConstraintFactory.cpp`. `create_boundary_event_constraint` parses the text of a specialized boundary constraint and "decides" which specialized boundary class should be used to instantiate a new specialized boundary constraint object based on the text. So, to make a new specialized boundary event accessible to users, new `else if` blocks must be added that look for *uniquely identifiable* text within the text of the new specialized boundary constraint. Frequently, constraint text is organized as (all on one line)

```
p<phase number>_<departure or arrival><- or +>_<constraint name>_
<lower bound><lower bound units>_<upper bound><upper bound units>
```

(See the examples in later sections.) `create_boundary_event_constraint` splits the entire string at each underscore and puts the results in the `std::vector ConstraintDefinitionCell`. The constraint name — or `ConstraintDefinitionCell[2]` — is almost always used to identify the constraint in `create_boundary_event_constraint`. Thus, it is important to give the new

constraint a unique name — and to instruct `create_boundary_event_constraint` to search for unique text!

Within the new `else if` block, the only action required is to `return` a `new` appropriate specialized boundary constraint object. The code from an existing block may therefore be copied into a new block, with the only required change being the name of the class derived from `SpecializedBoundaryConstraintBas`

Important note: All of the `if/else if` blocks based on `ConstraintDefinitionCell[2]` are contained within an outer `if/else if` block based on `EventDefinition`. `EventDefinition` can be either "departure" or "arrival". If the desired behavior is for the new constraint to be applicable at either a departure or arrival event, then an identical `else if` block must be added to both the `if` and `else` blocks of the `EventDefinition`-based block. Alternatively, if the desired behavior is for the new constraint to be applicable to only departure *or* arrival events, then the new code block should still be added to both sections, but should throw a helpful exception if the user attempts to invoke the constraint incorrectly (i.e., if the user tries to apply a departure-only constraint at an arrival event).

2.1.8 Updating the Executables

For some constraints, it may be necessary to update the contents of the executable files `EMTG_v9.cpp` and `testbed_driver.cpp`. The reason to update the executables is if the new constraint is allowed to depend on a universe body other than the central body of the journey.[2] If the constraint is allowed to depend on a universe body other than the central body, then the executables must be updated to create splines of the states of any additional universe bodies whose states are required by the constraint. To accomplish this, a new block of code must be placed in each executable inside the `for` loop that begins

```
//boundary constraint list
//some, but not all, require new splines
for (std::string& constraint : options.Journeys[j].BoundaryConstraintDefinitions)
{
...
}
```

The new block of code can be modeled after the existing `if` blocks inside this loop. For example, for the distance constraint, we have:

```
// distance constraint
if (boost::to_lower_copy(ConstraintDefinitionCell[2]).find("distanceconstraint") < 1024)
{
```

[2] An example of a constraint that does depend on an additional universe body is the RPR angle constraint because one or both of the reference bodies could be a universe body other than the central body. An example of a constraint that does not depend on an additional universe body is the semimajor axis constraint because the semimajor axis is always calculated relative to the central body of the constraint.

```
if (boost::to_lower_copy(ConstraintDefinitionCell[3]) != "cb")
{
int bodyIndex = std::stoi(ConstraintDefinitionCell[3]) - 1;

body_index_array.push_back(bodyIndex);
}
}
```

The outer `if` block determines whether a given boundary constraint is used. The inner `if` block checks to see if a universe body other than the central body is required to calculate the constraint, determines which universe body is required, and adds that body to the array of bodies to be splined. Note that the contents of the inner `if` block may change based on the individual constraint.

2.1.9 Tips and Tricks

Depending on what the constraint is on, it may be appropriate to scale the constraint. For example, constraints on distances are usually scaled by 1 LU.

It may be appropriate to allow the user to set units on the constraint.

Routines for obtaining existing state information:

- `BoundaryEventBase::get_state_before_or_after_event(int before_or_after)`
  - Returns state before event if `before_or_after` == -1; returns state after event if `before_or_after` == 1.

Routines for obtaining existing derivative information:

- `BoundaryEventBase::get_Derivatives_of_StateBeforeOrAfterEvent(int before_or_after)`
  - Returns state before event if `before_or_after` == -1; returns state after event if `before_or_after` == 1.
- `BoundaryEventBase::get_Derivatives_of_StateBeforeOrAfterEvent_wrt_Time(int before_or_after)`
  - Returns state before event if `before_or_after` == -1; returns state after event if `before_or_after` == 1.

Routines for computing orbit elements:

- `BoundaryEventBase::setComputeOrbitElements()`

- `BoundaryEventBase::add_orbit_element_reference_frame()`
- `BoundaryEventBase::get_orbit_elements_after_event()`
- `BoundaryEventBase::get_orbit_element_Jacobian_after_event()`

2.1.10 A Note on States and Derivatives Before and After Boundaries

Section 2.1.9 describes methods for obtaining state and derivative information infinitesimally before or after a boundary event. Some constraints, like the constraint on a state in a two-body rotating frame (Section 2.23), allow the user to use a `-` or `+` to indicate that they wish to constrain the state infinitesimally before or after a boundary event. However, it is important to realize that, while these methods exist, the backend EMTG data, unfortunately, does not always contain the information the user may want or expect when using these methods. The list below describes what EMTG actually stores for commonly used boundary types and classes. In the list, the following shorthand is used:

- $x-$: State obtained by `get_state_before_event()`.
- $x+$: State obtained by `get_state_after_event()`.
- x_1: State given in .emtg file for `Boundary: 1` for a journey.
- x_2: State given in .emtg file for `Boundary: 2` for a journey.
- $x_{e,1}$ State given in .emtg file in first row of mission events for a journey.
- $x_{e,-1}$ State given in .emtg file in last row of mission events for a journey.
- Δv_1, Δv_{-1} Values given in `dV_x`, `dV_y`, and `dV_z` columns of the first or last mission event in the .emtg file for a journey.

- Departure types/classes:
  - Type 0: Launch or direct insertion
    - Class 0: Ephemeris-pegged
      - $x- = x+ = x_{e,1}$
      - $v_{e,1} = v_1 + \Delta v_1$
      - Δv_1 is set by the decision variables for v_∞, RA, and DEC.
      - As currently implemented, I would consider both $x-$ and $x+$ to actually by $x+$ (i.e., after the launch or direct insertion maneuver).
    - Class 1: Free point
      - $x- = x+ = x_{e,1}$
      - $v_{e,1} = v_1 + \Delta v_1$
      - Δv_1 is set by the decision variables for v_∞, RA, and DEC.

            - As currently implemented, I would consider both $x-$ and $x+$ to actually by $x+$ (i.e., after the launch or direct insertion maneuver).
        * Class 3: Periapse
            - $x- = x+ = x_{e,1} = x_1$
            - Δv_1 is zero.
            - As currently implemented, I would consider both $x-$ and $x+$ to actually by $x+$ (i.e., after the launch or direct insertion maneuver).
    - Type 2: Free direct departure
        * Class 1: Free point
            - $x- = x+ = x_{e,1} = x_1$
            - Δv_1 is zero.
            - It makes sense that all position and velocity states are equal because there is no maneuver.
    - Type 3: Flyby
        * Class 0: Ephemeris-pegged
            - $x- = x+ = x_{e,1}$
            - $v_{e,1} = v_1+$ the journey's v_∞ decision variables.
            - As currently implemented, I would consider both $x-$ and $x+$ to actually by $x+$ (i.e., after the velocity change from the flyby has been imparted).

- Arrival types/classes:
    - Type 0: Insertion into parking orbit with chemical Isp
        * Class 0: Ephemeris-pegged
            - $x- = x+ = x_{e,-1}$
            - $v_{e,-1} = v_2 - \Delta v_{-1}$
            - $\Delta v_{-1} = -v_\infty$ from the decision variables.
            - It makes sense that $x- = x+$ even though v_2 is different because a maneuver is not actually being performed at the boundary. EMTG is just comparing the spacecraft state to the boundary state using v_∞.
    - Type 1: Rendezvous with chemical maneuver
        * Class 1: Free point
            - $x+ = x+ = x_{e,-1} = x_2$
            - $v- = (v+) - \Delta v_{-1}$
            - This one actually behaves out I would expect it to behave.
    - Type 2: Intercept with bounded v infinity
        * Class 0: Ephemeris-pegged
            - $x- = x+ = x_{e,-1}$
            - $v_{e,-1} = v_2 - \Delta v_{-1}$
            - $\Delta v_{-1} = -v_\infty$ from the decision variables.
            - It makes sense that $x- = x+$ even though v_2 is different because a maneuver is not actually being performed at the boundary. EMTG is just comparing the spacecraft state to the boundary state using v_∞.

* Class 1: Free point
  - $x- = x+ = x_{e,-1}$
  - $v_{e,-1} = v_2 - \Delta v_{-1}$
  - $\Delta v_{-1} = -v_\infty$ from the decision variables.
  - It makes sense that $x- = x+$ even though v_2 is different because a maneuver is not actually being performed at the boundary. EMTG is just comparing the spacecraft state to the boundary state using v_∞.
* Class 2: Ephemeris-referenced
  - $x- = x_{e,-1}$
  - $v_{e,-1} = v_2 - \Delta v_{-1}$, which is derived from the decision variables.
  - $x+ = x_{e,-1}$ BUT $x+$ is returned to the user relative to the boundary-event body, NOT the central body of the journey.
  - It makes sense that $x- = x+$, though expressed relative to different central bodies, even though v_2 is different because a maneuver is not actually being performed at the boundary. EMTG is just comparing the spacecraft state to the boundary state using v_∞.
* Class 3: Periapse
  - $x- = x+ = x_{e,-1} = x_2$
  - It makes sense that $x- = x+$ because a maneuver is not actually being performed at the boundary. EMTG is just comparing the spacecraft state to the boundary state using v_∞.

In summary, there are some boundaries at which it would be expected that the before/after states be the same. There are there boundaries at which it would be expected that the before/after states be different, but EMTG returns the same value. This may be changed in the future, but it has not been changed yet.

2.2 Classical Orbit Element Boundary Constraints

Distance-based constraints may be specified in kilometers (km), nautical miles (nmi), Astronomical Unit (AU) (AU) or universe length units (LU). Angular constraints may be specified in degrees (deg) or radians (rad). Time constraints may be specified in seconds (sec), hours (hr), days (d) or years (yr). The orientation COEs require that frame specification (e.g. ICRF, J2000BCI etc.).

2.2.1 Semimajor Axis

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_arrival_SMA_10000km_12000km
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.2.2 Inclination

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_departure_INC_40deg_40.1deg_J2000BCI
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.2.3 Eccentricity

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_arrival_ECC_0.6_0.7
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.2.4 Right Ascension of the Ascending Node

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_arrival_RAAN_0.1rad_0.15rad_ICRF
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.2.5 Argument of Periapse

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_arrival_AOP_0.1rad_0.15rad_TrueOfDateBCF
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.2.6 True Anomaly

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_arrival_trueanomaly_10deg_15deg_ICRF
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.3 COE Derived Constraints

2.3.1 Orbit Period

For closed orbits (i.e. $a > 0.0$) the spacecraft's orbital period T can be constrained:

$$T = \sqrt{\frac{a^3}{\mu}} \tag{1}$$

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_departure_orbitperiod_5.0d_5.3d
END_BOUNDARY_CONSTRAINT_BLOCK
```

In general, it is preferable to specify this constraint as an SMA constraint 2.2.1 in the event that the spacecraft state results in $a <= 0.0$ (which can happen during optimizer iteration).

2.3.2 Periapse Distance

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_arrival_periapsedistance_10000nmi_12000nmi
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.3.3 Apoapse Distance

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_arrival_apoapsedistance_10000nmi_12000nmi
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.4 Orbital Energy

Specific orbital energy ϵ can be constrained:

$$\epsilon = \frac{1}{2}v^2 - \frac{\mu}{r} \tag{2}$$

Currently, orbital energy can only be specified in units of km^2/s^2.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
pP_arrival_orbitalenergy_10.0km2s2_10.3km2s2
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.5 Distance Constraint

The user may constrain the distance between any boundary condition and any body in the Universe file. The user specifies the phase prefix, "arrival" or "departure," the body of interest as either "cb" or an integer index from the Universe file, and lower and upper bounds. The constraint may be posed in km, AU, or Universe LU.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_distanceconstraint_3_0.0au_1.0au
p0_departure_distanceconstraint_cb_0.0au_3.5au
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.6 Angle Constraints

The user may constrain the angle between the sun, the spacecraft, and a reference body. This may be done in terms of the Sun-Body-Probe (SBP) or Sun-Probe-Body (SPB). This set of constraints can be used to represent anything from the phase angle at encounter with an asteroid (Sun-asteroid-Probe) or the Sun-Earth-Probe (SEP) or Sun-Probe-Earth (SPE) angles.

There are four versions of these constraints as detailed below. The first two versions apply only to ephemeris-pegged boundary conditions that encode a v_∞ vector, such as intercept/flyby, chemical rendezvous, and orbit insertion. The last two may be applied to any class of boundary condition.

1. **Ephemeris-pegged Reference-Body-Probe (RBP)**. The user only needs to provide the reference body. The constraint uses the boundary position vector to represent the Body and the sum of the boundary position vector and the spacecraft v_∞ vector to represent the Probe. An example would be Sun-asteroid-Probe angle for an ephemeris-pegged asteroid flyby, where the Sun is the Reference and the asteroid is the Body.

2. **Ephemeris-pegged Reference-Probe-Body (RPB)**. The user only needs to provide the reference body. The constraint uses the boundary position vector to represent the Body and the sum of the boundary position vector and the spacecraft v_∞ vector to represent the Probe. An example would be Sun-Probe-asteroid angle for an ephemeris-pegged asteroid flyby, where the Sun is the Reference and the asteroid is the Body.

3. **Reference-Reference-Probe (RRP)**. The user specifies two reference bodies and the spacecraft's position vector is used for the Probe. One example is Sun-Earth-Probe angle, where the Sun and the Earth are the two references. Another example is Sun-asteroid-Probe angle for a flyby that is specified with any boundary class other than ephemeris-pegged.

4. **Reference-Probe-Reference (RPR)**. The user specifies two reference bodies and the spacecraft's position vector is used for the Probe. One example is Sun-Probe-Earth angle, where the Sun and the Earth are the two references. Another example is Sun-Probe-asteroid angle for a flyby that is specified with any boundary class other than ephemeris-pegged.

2.6.1 RBP and RPB angle constraints

The RBP and RPB constraints are specified as shown below. The user may identify the reference body using either "cb" for the central body or an integer encoding a body's index in that Journey's Universe file. The lower and upper bounds on the constraint are specified in degrees.

The RBP and RPB constraints may only be specified for ephemeris-pegged arrival events that encode a v_∞ vector, *i.e.* intercept/flyby, chemical rendezvous, and orbit insertion. Note that because RBP and RPB only work with ephemeris-pegged boundary event, we can make the assumption that the "body" is the body associated with the boundary event and so we only need to encode the "reference" in the constraint description as shown below.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_RBP_cb_0.0_15.0
p3_arrival_RPB_5_8.3_17.2
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.6.2 RRP and RPR angle constraints

The RRP and RPR constraints are specified as shown below. The syntax for RRP and RPR requires that the user specify two reference bodies. The user may identify the reference body using either "cb" for the central body or an integer encoding a body's index in that Journey's Universe file. The lower and upper bounds on the constraint are specified in degrees.

The RRP and RPR constraints may be applied to any boundary event.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_RRP_cb_3_0.0_15.0
p3_departure_RPR_5_7_8.3_17.2
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.7 Angular Momentum Reference Angle Constraint

The user may choose to constrain the angle between the spacecraft's ICRF angular momentum vector relative to the central body and the vector to some reference body. For example, the user may need to constrain that the final orbit of the spacecraft about some body be in the terminator plane, *i.e.* the angle between the vector to the sun and the angular momentum vector is zero.

The user may supply this constraint in degrees or radians.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_angularMomentumReferenceAngle_cb_0.0deg_15.0deg
```

```
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.8 Departure or Arrival Maneuver Thruster Set

The user may direct a particular chemical departure or arrival maneuver to be either "monoprop" or "biprop." EMTG will then use the I_{sp} specified for monoprop or biprop in the spacecraft file.

This class of constraint is applied to departure type `0: launch or direct insertion` (when a launch vehicle is not used) and arrival types `0: insertion into parking orbit (use chemical Isp)` and `1: rendezvous (with chemical maneuver)`, as well as powered patched-conic flybys. All other boundary types will ignore this class of constraint.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_monoprop
p3_departure_biprop
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.9 Longitude

The user may constrain longitude in ICRF, J2000BCF or TrueOfDateBCF. Longitude is computed as,

$$\lambda = \text{atan2}\left(y, x\right) \tag{3}$$

and is given in the range $[-180.0, 180.0]$. The user provides lower and upper bounds on longitude, in degrees.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_departure_longitude_84.0_85.0
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.10 BCF Latitude

The user may constrain latitude in the J2000 body-centered fixed (BCF) frame as defined in the current journey's Universe file. This is also known as geocentric latitude as opposed to geodetic latitude. Latitude is computed as,

$$\begin{aligned} \theta &= \text{atan2}\left(z_{J2000BCF}, r_{xy_{J2000BCF}}\right) \\ r_{xy_{J2000BCF}} &= \sqrt{\left(x_{J2000BCF}^2 + y_{J2000BCF}^2\right)} \end{aligned} \tag{4}$$

and is given in the range $[-180.0, 180.0]$. The user provides lower and upper bounds on longitude, in degrees.

Note that this constraint is computed in the *universe's central body's* J2000 BCF frame. This is true regardless of boundary class.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_departure_BCFlatitude_-29.0_-28.0
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.11 (Geo)Detic Latitude

The user may constrain (geo)detic latitude. The user may specify this constraint in J2000BCF and TrueOfDateBCI frames. The user provides lower and upper bounds on latitude, in degrees. Note that this constraint has a singularity at the pole.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_DeticLatitude_25.0_26.0_trueofdatebcf
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.12 (Geo)Detic Altitude

The user may constrain (geo)detic altitude. The user may specify this constraint in J2000BCF and TrueOfDateBCI frames. The user provides lower and upper bounds on altitude, in km. Note that (derivative of) this constraint has a singularity at the pole.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_DeticAltitude_206.0_207.0_trueofdatebcf
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.13 (Geo)Detic Target Body Elevation

The user may constrain the (geo)detic elevation of a target body with respect to the surface normal vector of a spheroid with flattening factor f. The target body is specified using an integer encoding a body's index in that Journey's Universe file. The user may specify this constraint in J2000BCF and TrueOfDateBCI frames. The user provides lower and upper bounds on detic elevation, in degrees.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_TargetDeticElevation_10_70.0_90.0_trueofdatebcf
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.14 (Geo)Detic Elevation from Ground Station

The user may constrain the (geo)detic elevation of the spacecraft with respect to a ground station. The location of the ground station is defined using latitude/longitude/altitude in a true-of-date, body-fixed reference frame relative to the body on which the ground station is located.

The syntax of the constraint following the `DeticElevationFromGroundStation` text block is:

1. ID for body on which the ground station is located. Either `CB` for central body or a number for the body ID.
2. Latitude of ground station.
3. Longitude of ground station.
4. Altitude of ground station.
5. Lower bound on elevation angle.
6. Upper bound on elevation angle.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_DeticElevationFromGroundStation_CB_28.455deg_-52.246deg_0.0km_60.0deg_60.0deg
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.15 Velocity Declination

The user may constrain velocity declination, given by:

$$\delta_v = \text{atan2}\left(v_z, v_{xy}\right) \tag{5}$$

where v_{xy} is the projection of the velocity vector onto the x-y plane. The user may specify this constraint in ICRF, J2000BCI, J2000BCF, TrueOfDateBCI, and TrueOfDateBCF frames.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_departure_VelocityDeclination_-20.0_5.0_ICRF
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.16 Velocity Declination with Respect to Any Body

The velocity declination constraint described in Section 2.15 can only be used to constrain the velocity with respect to the central body of the journey. The constraint described here is the

same mathematical construct, but can be referenced to any body defined in the relevant journey's universe.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p<phase number>_<boundary><before or after boundary>_VelocityDeclinationAnyBody_
<Body ID>_<lower bound><lower bound units>_<upper bound><upper bound units>_<Frame>
END_BOUNDARY_CONSTRAINT_BLOCK
```

where

- `<phase number>` is the integer number of the phase on which the constraint is applied.
- `<boundary>` is `departure` or `arrival`.
- `<before or after boundary>` is `-` or `+`. A `-` indicates that the constraint is applied infinitesimally before the boundary. A `+` indicates that the constraint is applied infinitesimally after the boundary. See Section 2.1.10 for a full description of what `-` or `+` actually means for a given boundary type/class.
- `<body ID>` is the ID for the body with respect to which the velocity declination is calculated. The velocity vector of the spacecraft is calculated with respect to this body, and the reference frame with respect to which the declination angle of the velocity vector is calculated is defined by this body. May be either `cb` to indicate the central body of the journey or any numerical body ID defined in the journey's universe file body list.
- `<lower bound>` The numerical lower bound for the constraint.
- `<lower bound units>` The units of the lower bound. May be `deg` or `rad`.
- `<upper bound>` The numerical upper bound for the constraint.
- `<upper bound units>` The units of the upper bound. May be `deg` or `rad`.
- `<Frame>` The reference frame with respect to which the declination angle is calculated, defined by `body ID`. Valid choices are `ICRF`, `J2000BCI`, `J2000BCF`, `TrueOfDateBCI`, and `TrueOfDateBCF`.

Syntax examples:

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_departure-_VelocityDeclinationAnyBody_CB_9_-1.0rad_2rad_ICRF
p0_arrival+_VelocityDeclinationAnyBody_3_10deg_50deg_TrueOfDateBCF
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.17 Vertical Flight Path Angle (VFPA)

The user may constrain vertical flight path angle, given by:

$$\psi = \arccos\left(\frac{\mathbf{r}\cdot\mathbf{v}}{rv}\right) \tag{6}$$

where the CF denotes the constraint frame. The user may specify this constraint in ICRF, J2000BCI, J2000BCF, TrueOfDateBCI, and TrueOfDateBCF frames.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_VFPA_97.0_98.0_J2000BCF
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.18 Velocity magnitude

The user may constrain velocity magnitude in km/s. The user may specify this constraint in ICRF, J2000BCI, J2000BCF, TrueOfDateBCI, and TrueOfDateBCF frames.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_VelocityMagnitude_11.9_12.1_J2000BCF
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.19 Velocity Spherical Azimuth

The user may constrain spherical azimuth angle of inertial velocity vector in degrees. The spherical azimuth angle is given as

$$Az = \arctan\frac{v_{east}}{v_{north}} \tag{7}$$

,where v_{east} and v_{north} are inertial velocity elements at the spherical local coordinate as

$$\begin{bmatrix} v_{up} \\ v_{east} \\ v_{north} \end{bmatrix} = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\lambda & \sin\lambda & 0 \\ -\sin\lambda & \cos\lambda & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_r \\ v_r \\ v_r \end{bmatrix} \tag{8}$$

Here, ϕ is a (geo)centric latitude and λ is a longitude. The user may specify this constraint in J2000BCF and TrueOfDateBCF frames.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_VelocitySphericalAzimuth_50.1_50.2_trueofdatebcf
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.20 Relative Velocity magnitude

The user may constrain magnitude of velocity vector relative to central body's ground (or atmosphere) in km/s. The relative velocity vector and the magnitude are given by

$$\vec{v_r} = \vec{v} - \vec{\omega} \times \vec{r} = \vec{v} - \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \vec{r} \tag{9}$$

$$v_r = |\vec{v_r}| = \sqrt{v_{rx}^2 + v_{ry}^2 + v_{rz}^2} \tag{10}$$

The user may specify this constraint in J2000BCF and TrueOfDateBCF frames.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_RelativeVMagnitude_11.9_12.1_J2000BCF
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.21 Relative Velocity Azimuth

The user may constrain azimuth angle of velocity vector relative to central body's ground (or atmosphere) in degrees. The azimuth angle is given as

$$Az = \arctan \frac{v_{r_{east}}}{v_{r_{north}}} \tag{11}$$

,where $v_{r_{east}}$ and $v_{r_{north}}$ are relative velocity elements at the horizontal local coordinate as

$$\begin{bmatrix} v_{r_{up}} \\ v_{r_{east}} \\ v_{r_{north}} \end{bmatrix} = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\lambda & \sin\lambda & 0 \\ -\sin\lambda & \cos\lambda & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_{rx} \\ v_{ry} \\ v_{rz} \end{bmatrix} \tag{12}$$

Here, ϕ is a (geo)detic latitude and λ is a longitude. The user may specify this constraint in J2000BCF and TrueOfDateBCF frames.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_RelativeVAzimuth_50.1_50.2_trueofdatebcf
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.22 Relative Velocity Horizontal Flight Path Angle(HFPA)

The user may constrain horizontal flight path angle of velocity vector relative to central body's ground (or atmosphere) in degrees. The HFPA is given as

$$HFPA = \arctan \frac{v_{r_{up}}}{\sqrt{v_{r_{north}}^2 + v_{r_{east}}^2}} \tag{13}$$

,where $v_{r_{east}}$ and $v_{r_{north}}$ are relative velocity elements at the horizontal local coordinate as

$$\begin{bmatrix} v_{r_{up}} \\ v_{r_{east}} \\ v_{r_{north}} \end{bmatrix} = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix} \begin{bmatrix} \cos\lambda & \sin\lambda & 0 \\ -\sin\lambda & \cos\lambda & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_{rx} \\ v_{ry} \\ v_{rz} \end{bmatrix} \tag{14}$$

Here, ϕ is a (geo)detic latitude and λ is a longitude. The user may specify this constraint in J2000BCF and TrueOfDateBCF frames.

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_arrival_RelativeVHFPA_-14.1_-14.0_trueofdatebcf
END_BOUNDARY_CONSTRAINT_BLOCK
```

2.23 State in Two-Body Rotating Frame

The user may constrain the state of the spacecraft relative to a two-body rotating reference frame. This frame is analogous to, e.g., GMAT's "ObjectReferenced" frame type. The user sets two bodies in the journey's universe to define the axes of the frame. The x axis points from the first body to the second body, the z axis points in the direction of the angular momentum vector of the second body's motion about the first body, and the y axis completes the right-handed set. Notes on implementation:

- The user may constrain independently any element of the position/velocity state of the spacecraft in this frame.
- The constraint may be applied at arrival or departure of a phase. Additionally, the constraint may be applied infinitesimally before or after the phase boundary time in order to take into account (or not) events that happen instantaneously at a boundary (e.g., an impulsive maneuver).
- Currently, the constrained state can only be defined relative to the second body used to define the frame.
- Position constraints are expressed in the rotating frame.
- Velocity constraints are calculated by differentiating in the rotating frame *and* expressing the velocity in the rotating frame.

Syntax (all one line):

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p<phase number>_<boundary><before or after boundary>_StateInTwoBodyRotatingFrame
_<state element>_<body ID for B1>_<body ID for B2>_<origin of frame>
_<lower bound><lower bound units>_<upper bound><upper bound units>
END_BOUNDARY_CONSTRAINT_BLOCK
```

where

- `<phase number>` is the integer number of the phase on which the constraint is applied.

- `<boundary>` is `departure` or `arrival`.
- `<before or after boundary>` is `-` or `+`. A `-` indicates that the constraint is applied infinitesimally before the boundary. A `+` indicates that the constraint is applied infinitesimally after the boundary. See Section 2.1.10 for a full description of what `-` or `+` actually means for a given boundary type/class.
- `<state element>` is the state to be constraint. Valid values are: `x`, `y`, `z`, `vx`, `vy`, `vz`.
- `<body ID for B1>` is the ID for the first body that defines the frame. May be either `cb` to indicate the central body of the journey or any numerical body ID defined in the journey's universe file body list. Cannot be the same as `<body ID for B2>`.
- `<body ID for B2>` is the ID for the second body that defines the frame. May be either `cb` to indicate the central body of the journey or any numerical body ID defined in the journey's universe file body list. `<body ID for B1>`.
- `<origin of frame>` Defines the origin of the rotating frame. Currently, the only valid value is `body2`.
- `<lower bound>` The numerical lower bound for the constraint.
- `<lower bound units>` The units of the lower bound. If `<state element>` is `x`, `y`, or `z`, then valid values are `km`, `AU`, or `LU`. If `<state element>` is `vx`, `vy`, or `vz`, then valid values are `km/s`.
- `<upper bound>` The numerical upper bound for the constraint.
- `<upper bound units>` The units of the upper bound. If `<state element>` is `x`, `y`, or `z`, then valid values are `km`, `AU`, or `LU`. If `<state element>` is `vx`, `vy`, or `vz`, then valid values are `km/s`.

Syntax examples:

```
BEGIN_BOUNDARY_CONSTRAINT_BLOCK
p0_departure-_StateInTwoBodyRotatingFrame_VY_CB_9_Body2_-1km/s_2km/s
p0_arrival+_StateInTwoBodyRotatingFrame_X_7_9_Body2_1000km_2000km
END_BOUNDARY_CONSTRAINT_BLOCK
```

Additional documentation is available in the directory `docs/constraint_in_two_body_rotating_frame`. The PowerPoint presentation `EMTG-182_Constraint_In_Rotating_Frame.pptx` describes the basic architectural design of the constraint. `constraint_in_two_body_rotating_frame.pdf` gives the full mathematical specification.

3 MGAnDSMs Maneuver Constraints

The multiple gravity assist with n deep-space maneuvers using shooting (MGAnDSMs) transcription allows the optimizer to place up to n deep-space maneuvers in each phase of a multiple

gravity-assist mission. Nominally, the numerical optimizer will place the maneuvers in locally optimal locations based on the nonlinear program (NLP) necessary conditions for optimality. There are, however, many instances where the mission designer will want to constrain the parameters of a particular maneuver for practical reasons including, but not limited to:

1. Navigation restrictions: no maneuver pre/post another mission critical event
2. Engine type: restrict a certain maneuver to be executed with a particular propulsion system (bipropellant or monopropellant thruster sets)
3. Magnitude: constrain the size of a particular maneuver due to hardware limitations or practical navigation concerns

Individual maneuver constraints are specified in the .emtgopt file. Each journey has a maneuver constraint block:

```
BEGIN_MANEUVER_CONSTRAINT_BLOCK
constraint_1
constraint_2
constraint_3
.
.
.
constraint_n
END_MANEUVER_CONSTRAINT_BLOCK
```

The basic maneuver constraint syntax identifies the journey and phase that the maneuver is in, as well as its index within the phase (indexed from zero).

Instructions on how to specify different types of maneuver constraints are provided in the following sections along with examples.

3.1 Maneuver Epoch

Maneuver positions can be specified via their epoch in a variety of ways: an absolute epoch, epoch offset with respect to another mission event (e.g. another maneuver's epoch), or an epoch offset w.r.t. a right or left phase boundary.

3.1.1 Absolute Epoch

A maneuver may be fixed in time by specifying either a Modified Julian Date (MJD) or Julian Date (JD).

```
BEGIN_MANEUVER_CONSTRAINT_BLOCK
p0b0_epoch_abs_51544.0_51544.5
END_MANEUVER_CONSTRAINT_BLOCK
```

3.1.2 Epoch Relative to Phase Boundary

A maneuver may be defined relative to the left or right boundary of the phase, in units of days.

```
BEGIN_MANEUVER_CONSTRAINT_BLOCK
p0b0_epoch_lboundary_14.0_10000.0
p0b0_epoch_rboundary_60.0_10000.0
END_MANEUVER_CONSTRAINT_BLOCK
```

3.1.3 Epoch Relative to Previous or Next Event

A maneuver may be defined relative to the previous or next event in the phase (*i.e.* another maneuver or a boundary), in units of days.

```
BEGIN_MANEUVER_CONSTRAINT_BLOCK
p0b0_epoch_next_14.0_10000.0
p0b0_epoch_previous_60.0_10000.0
END_MANEUVER_CONSTRAINT_BLOCK
```

3.2 Maneuver Magnitude

The user may constrain the magnitude of any MGAnDSMs maneuver, in units of km/s.

```
BEGIN_MANEUVER_CONSTRAINT_BLOCK
p0b0_magnitude_0.0_0.01
END_MANEUVER_CONSTRAINT_BLOCK
```

3.3 Maneuver Thruster Set

The user may direct a particular MGAnDSMs maneuver to be either "monoprop" or "biprop." EMTG will then use the I_{sp} specified for monoprop or biprop in the spacecraft file.

```
BEGIN_MANEUVER_CONSTRAINT_BLOCK
p0b0_monoprop
END_MANEUVER_CONSTRAINT_BLOCK
```

4 Parallel Shooting Maneuver Constraints

The parallel shooting transcriptions, Parallel Shooting with Finite-Burn (PSFB) and Parallel Shooting with Bounded Impulses (PSBI) transcriptions decompose a low-thrust phase into n subphases, or 'steps." Each individual step may have up to m maneuvers, or "substeps," each of which has its own NLP control parameters. As in MGAnDSMs, constraints may be placed in the maneuver constraint block. PSFB and PSBI are designed such that constraints are placed at the step level, not at the substep level.

4.1 Parallel Shooting Duty Cycle

The user may directly constrain the duty cycle in each parallel shooting step as a floating point variable in [0.0, 1.0]. If EMTG is set to "averaged" duty cycle mode, this duty cycle will be applied as a multiplier to both thrust and mass flow rate. If EMTG is set to "high-fidelity" duty cycle mode, then the spacecraft will thrust for that percentage of the step length and then insert a forced coast until the end of the step.

```
BEGIN_MANEUVER_CONSTRAINT_BLOCK
p0b0_dutycycle_0.9
END_MANEUVER_CONSTRAINT_BLOCK
```

4.2 Parallel Shooting body-probe-thrust angle

The user may constrain the body-probe-thrust (BPT) angle for any PSFB or PSBI phase of the CAESAR trajectory. This constraint applies to an entire phase rather than a particular step of that phase. For convergence reasons, the constraint is on the cosine of the BPT angle rather than the BPT angle itself.

The upper and lower bounds on $\cos(\theta_{BPT})$ are expressed as functions of r, the distance between the spacecraft and a reference body of the user's choice. This formulation allows for BPT angle constraints that become stricter as the spacecraft flies closer to the reference body and less strict as the spacecraft flies away from the reference body. The upper and lower bounds are fit using the four parameter logistic (4PL) curve as per Equations 15.

$$\begin{aligned} L(r) \leq \cos(\theta_{BPT}) \leq U(r) \\ U(r) = \alpha_{Ud} + \frac{\alpha_{Ua} - \alpha_{Ud}}{1.0 + \left(\frac{r}{\alpha_{Uc}}\right)^{\alpha_{Ub}}} \\ L(r) = \alpha_{Ld} + \frac{\alpha_{La} - \alpha_{Ld}}{1.0 + \left(\frac{r}{\alpha_{Lc}}\right)^{\alpha_{Lb}}} \end{aligned} \tag{15}$$

When specifying the constraint, the user must provide four coefficients each for the 4PL curves

associated with the maximum and minimum values for $\cos(\theta_{BPT})$, as well as the identity of the reference body. The reference body may be encoded as "cb" for the central body, or with an integer code describing the reference body's position in the Universe file. Note that the blow code block needs to be one line in the .emtgopt file.

Alternatively, the user may specify the constraint with a constant lower and upper bound. The bounds are given in degrees.

```
BEGIN_MANEUVER_CONSTRAINT_BLOCK
p0_bpt_cb_0.1879254,24.6087500,0.9480499,0.8878154...
           _1.6239600,14.9903500,0.8006161,-0.8878154
p0_bpt_cb_27.4_152.6
END_MANEUVER_CONSTRAINT_BLOCK
```

5 Phase Distance Constraint

The user may constrain the distance between the spacecraft and any other body in the universe file at each time at which a maneuver occurs. The body may be expressed either as "cb" for the central body, or with an integer code describing the reference body's position in the Universe file. The lower and upper bounds on the distance constraint may be specified in LU, AU, or km.

At the moment, this constraint works in multiple gravity assist with low-thrust (MGALT), finite-burn low-thrust (FBLT), PSBI, PSFB, and MGAnDSMs. Note that, for MGAnDSMs especially (because its maneuvers are potentially spaced so far apart), this constraint does not necessarily mean that the spacecraft never violates this constraint. It means that no point at which the spacecraft performs a maneuver violates this constraint.

Each Journey block in the .emtgopt file has its own phase distance constraint section.

```
BEGIN_PHASE_DISTANCE_CONSTRAINT_BLOCK
p0_4_0.9au_10.0au
END_PHASE_DISTANCE_CONSTRAINT_BLOCK
```

6 PyEMTG Interface

Scripted constraints may be accessed via PyEMTG through the following class properties.

6.1 Boundary Constraints

Boundary constraint definitions are stored on a per-journey basis as a list in

`JourneyOptions.BoundaryConstraintDefinitions`

Each element of the list is a string that contains the entirety of the boundary constraint text. In other words, the upper and lower bound, etc. of the constraint are not stored separately. If there are no boundary constraints for a given journey, then the list is empty.

Boundary constraint output is stored on a per-journey basis as a list in

`Journey.BoundaryConstraintOutputs`

Each element of the list is a string that contains the entirety of the boundary constraint output text (i.e., the entirety of the line written to the .emtg output file for that constraint). The user may then employ string-parsing techniques to, e.g., extract the numerical value of the constraint. If there are no boundary constraints for a given journey, then the list is empty.

6.2 Maneuver Constraints

Maneuver constraint definitions are stored on a per-journey basis as a list in

`JourneyOptions.ManeuverConstraintDefinitions`

Each element of the list is a string that contains the entirety of the maneuver constraint text. In other words, the upper and lower bound, etc. of the constraint are not stored separately. If there are no maneuver constraints for a given journey, then the list is empty.

6.3 Phase Distance Constraints

Phase distance constraint definitions are stored on a per-journey basis as a list in

`JourneyOptions.PhaseDistanceConstraintDefinitions`

Each element of the list is a string that contains the entirety of the phase distance constraint text. In other words, the upper and lower bound, etc. of the constraint are not stored separately. If there are no phase distance constraints for a given journey, then the list is empty.