EMTGv9 Software Design Document

Jacob A. Englander [1], Donald H. Ellison [2], Kyle Hughes [3],
Alec Mudek [4], Sean Napier [5], Bruno Victorino Sarli [6], Noble Hatten [7], Jeremy Knittel [8]

[1]EMTG product development lead, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

[2]EMTG development team lead, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

[3]EMTG test team lead, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

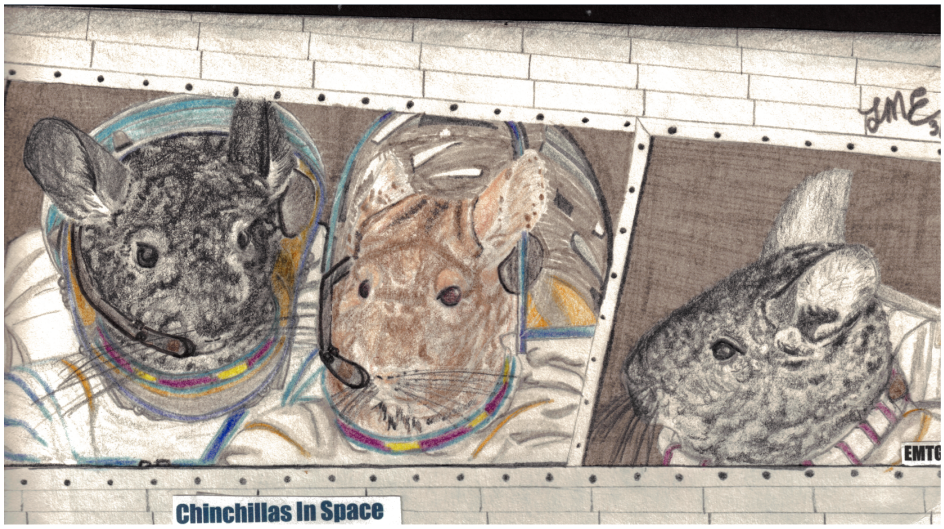
[4]EMTG test team member, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

[5]EMTG test team member, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

[6]EMTG test team member, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

[7]EMTG development team member, NASA Goddard Space Flight Center, Flight Dynamics and Mission Design Branch Code 595

[8]EMTG development team member emeritus



Revision Date	Author	Description of Change
eventually	us	first completed draft

Contents

List of Figures

List of Tables

List of Acronyms

ACS attitude control system

DLA Declination of Launch Asymptote

RLA Right Ascension of Launch Asymptote

RA right ascension

DEC declination

EMTG Evolutionary Mission Trajectory Generator

SNOPT Sparse Nonlinear OPTimizer

SOI sphere of influence

SQP sequential quadratic programming

IAU International Astronomical Union

ICRF International Celestial Reference Frame

MJD Modified Julian Date

SPICE Spacecraft Planet Instrument Camera-matrix Events

NLP nonlinear program

MBH monotonic basin hopping

MGALT Multiple Gravity Assist with Low-Thrust

MGALTS Multiple Gravity Assist with Low-Thrust using the Sundman transformation

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

FBLTS Finite-Burn Low-Thrust using the Sundman transformation

$\mathbf{I}_{sp}$ specific impulse

C_3 hyperbolic excess energy

GMAT General Mission Analysis Toolkit

PPU power processing unit

STM state transition matrix

MTM maneuver transition matrix

HPTM half-phase transition matrix

DSM deep-space maneuver

COE classical orbit elements

GSL Gnu Scientific Library

NEXT NASA's Evolutionary Xenon Thruster

SMA semi-major axis

ECC eccentricity

GSAD Ghosh Sparse Algorithmic Differentiation

Chapter 1

Introduction

This document presents a conceptual overview of the Evolutionary Mission Trajectory Generator (EMTG) design. Each functional unit of the program is described in text. This document is *not* intended to be a mathematics specification.

This document is intended to be used side-by-side with the EMTG Doxygen-generated documentation. The Software Design Document describes the functional units of the design, and the Doxygen output maps the design to the code.

Chapter 2

Common Design Elements

2.1 Internal Calculation Frame

All internal calculations in EMTG are performed in the International Celestial Reference Frame (ICRF) as defined in International Celestial Reference Frame (ICRF) as defined in the International Astronomical Union (IAU) Cartographic Coordinates and Rotational Elements document [1] and in Spacecraft Planet Instrument Camera-matrix Events (SPICE) [2]. SPICEś `J2000` frame is identical to ICRF. The frame is always centered on whatever celestial body or barycenter that is relevant to the calculation, *i.e.* the Sun, the Earth, or any other body or point *that can be defined in SPICE*.

In some cases, the user may specify states or constraints in frames other than ICRF, or request output in frames other than ICRF. EMTG handles state input and output in alternative frames by rotating into ICRF, performing the calculations, and then rotating back to the requested output frame only at the last moment before the output is printed. Constraints, as described in Sections 13.9, 12.2.3.1, and 12.2.3.1, work differently in that the constraint objects rotate EMTG's internal state into the frame of the constraint. The latter case occurs only in the individual constraint objects and therefore the rotated states are never available to the rest of the program and there is no opportunity for confusion as to what frame EMTG is working in.

2.2 Units

All internal calculations are performed in units of kilometers, kilograms, and seconds. If particular piece of code needs to use a different unit system, then conversion is performed at the point where it is necessary and not passed to other parts of the code. The most frequent use of non-standard units is in input and output functions.

_models/c-hardware_models.aux

Chapter 3

Core

3.1 EMTG enums

`EMTG_enums.h` is where we keep the enums that define EMTG's various modes. These enums are used to control switches in various class constructors throughout the rest of EMTG. The enums are listed below.

The following enums define the structure of the problem:

- **PhaseType** - allows the user to specify a phase type as defined in Chapter 12.
- **StateRepresentation** - allows the user to specify how a given state vector is described in the problem. This is used in parallel shooting transcriptions (Section 12.3) and in PeriapseBoundary and FreePointBoundary (Sections 13.7 and 13.8)
- **ObjectiveFunctionType** - defines the objective function type as defined in Chapter 14.

The following enums define the spacecraft:

- **SpacecraftModelInputType** - defines whether EMTG reads the spacecraft model from a file, constructs it from library entries, or constructs it from entries in the `.emtgopt` file, as described in Chapter 5.
- **SpacecraftBusPowerType** - defines the spacecraft bus power model as described in Chapter 5.
- **SpacecraftPowerSupplyType** - defines the spacecraft power supply type (solar vs radioisotope/nuclear) as described in Chapter 5.
- **SpacecraftPowerSupplyCurveType** - defines the spacecraft power supply curve type as described in Chapter 5.

- **SpacecraftThrusterMode** - defines which type of model to use for a given thruster, as described in Chapter 5.
- **ThrottleLogic** - defines which set of rules to use for determining the number of thrusters to fire, as described in Chapter 5.
- **PropulsionSystemChoice** - determines which propulsion system (monoprop, biprop, or electric) the spacecraft uses for a given maneuver, as described in Chapter 5.

The following enums define boundary events:

- **BoundaryClass** - defines whether a given boundary event will be constructed as an `EphemerisPeggedBoundary`, `FreePointBoundary`, `EphemerisReferencedBoundary`, or `PeriapseBoundary`, as described in Chapter 13.
- **DepartureType** - defines the type of a departure event, within its class, as described in Chapter 13.
- **ArrivalType** - defines the type of a arrival event, within its class, as described in Chapter 13.

The following enums control EMTG's physics engine:

- **PropagatorType** - defines whether EMTG uses a Kepler or integrated propagator for a given propagation as described in Chapter 6.
- **IntegratorType** - defines whether EMTG uses a fixed-step or adaptive-step integrator for a given propagation as described in Chapter 7.
- **PropagationDomain** - defines whether EMTG uses time-domain or Sundman-domain integration for a given propagation as described in Chapter 7.
- **DutyCycleType** - defines whether EMTG uses averaged or "realistic" duty cycle for thrust arcs in the PSFB transcription, as described in Section 12.3.1.
- **ReferenceFrame** - defines the reference frame for a given calculation, as described in Section 4.4.

The following enums control EMTG's solvers:

- **InnerLoopSolverType** - defines which inner-loop solver (NLP-only, MBH, textitetc.) will be used for this run of EMTG, as described in Chapter 15.
- **NLPMode** - defines whether EMTG runs its NLP solver in feasible point, optimize, or filament finder mode as described in Chapter 15.

3.2 doubleType

`doubleType.h` is a very simple header that can be included in any EMTG source file and enables the use of the Ghosh Sparse Algorithmic Differentiation (GSAD) package. It checks to see if the `AD_INSTRUMENTATION` compiler macro is set. If so, the `doubleType` macro is defined as `GSAD::adouble` and the `_getValue` and `_setValue(x, y)` macros are defined as `GSAD::adouble::getValue()` and `GSAD::adouble::setValue(x, y)` macros, respectively. If the `AD_INSTRUMENTATION` compiler macro is *not* set, then `doubleType` is defined as `double` and `_getValue` and `_setValue(x, y)` are defined as white space.

`doubleType.h`, while only 20 lines long, is the key piece of code that enables EMTG to be its own partial derivative checker.

3.3 Chinchilla

`Chinchilla.h` is a splash screen that draws an ascii picture of Clementine, the mascot of the EMTG program.

3.4 MissionOptions

The `MissionOptions` class reads and write the `.emtgopt` files that control EMTG. `MissionOptions` contains options that globally control EMTG, plus a vector of `JourneyOptions` objects that control each individual journey.

Because `MissionOptions` must be changed in several different places each time a new option is added, we chose to generated it using an auto-coder. The developer need only modify `OptionsOverhaul/list_of_missionoptions.csv` and then run *PyEMTG/OptionsOverhaul/make_EMTG_missionoptions_journeyoptions.py*. The auto-coder will then validate the .csv file to ensure that all necessary fields are present, and will then generate the `MissionOptions` and `JourneyOptions` classes in both C++ and Python. The fields of `MissionOptions` and `JourneyOptions` are defined in a .csv file because it is plain text and easy to configuration-manage via Git.

By default, `MissionOptions` will only write out a given option if it *does not match its default value*. This way each `.emtgopt` file is as short and easy to read as possible.

3.5 JourneyOptions

The `JourneyOptions` class controls the construction and evaluation of each `Journey` object and owned `Phase` objects. Like `MissionOptions`, `JourneyOptions` is generated via an auto-coder. The

same auto-coder generates both classes. Also like `MissionOptions`, `JourneyOptions` only writes out options that do *not* take their default values.

3.6 Problem

The `Problem` class is a base class for all EMTG problem types. As of this writing, `mission` (Chapter 10) is the only derived class of `Problem` but we keep the `Problem` class in case this changes later.

The main purpose of the `Problem` class is to hold all of the fields necessary to interact with an nonlinear program (NLP) problem solver, including the decision vector, constraint vector, bounds, Jacobian, and Jacobian sparsity pattern. In addition, `Problem` contains helper methods to check the infinity-norm of the constraint violation vector and to check the decision vector, constraint vector, and Jacobian for infinite or undefined values.

Chapter 4

Astrodynamics

Donald

4.1 Universe

The `Universe` class is the central storage class for all celestial body and ephemeris data required by EMTG. EMTG creates a single `Universe` object for each `Journey` in the EMTG mission shortly after the EMTG options file is parsed. An emtg_universe file is passed to the `universe` constructor and parsed using the `load_universe_data` method. If `SplineEphem` 8 is being used, then a `SplineEphem_universe` is also created and passed to the `Universe` constructor so that a spline-smoothed ephemeris may be used. The `universe` constructor also creates a `CentralBody` object that corresponds to the central body specified in the emtg_universe file. The `universe` objects that correspond to the `Journeys` in an EMTG mission are connected together in a singly-linked list, the current `universe` has a pointer to the next `universe` in the mission to facilitate SOI transitions across `EphemerisReferenced` boundary events. The `next_universe` pointer is accessed via the `get_nextUniverse()` method.

An EMTG `universe` mostly contains astrodynamics data pertaining to the `CentralBody` (4.3), whose geometric center serves as the center of propagation in EMTG. The `locate_central_body` method computes the Cartesian state of the `CentralBody` with respect to the Sun, and returns zero, if the `CentralBody` is the Sun. A `universe` also contains a vector of `Body` objects for each of the perturbation/flyby targets available in a .emtg_universe file. A `Frame` object is constructed using the `CentralBody` IAU Euler rotation angles (4.4) and their first time derivatives $\{\alpha, \dot{\alpha}, \delta, \dot{\delta}, W, \dot{W}\}$ that are specified in a .emtg_universe file. Gas drag on a spacecraft due to the atmosphere of the `CentralBody` can be modeled as discussed in section 4.8.5. To facilitate this, if it is required, an `atmosphere` object is constructed for each `universe` object that requires one. This is done shortly after the `universes` are assembled in the main `EMTG_v9.cpp` execution method.

The `universe` also contains some of the scaling information used by EMTG when it assembles the NLP problem passed to the NLP solver. The `continuity_constraint_scale_factors` con-

tainer contains multiplicative scaling factors that are applied to all trajectory defect constraints that appear in EMTG's trajectory transcriptions. Position defects are multiplied by $1/LU$, where LU is specified by the user in the emtg_universe file. A time scaling unit TU is computed using the `CentralBody` μ and the provided LU. Velocity defects are multiplied by TU/LU. Mass defects are multiplied by $1/MU$ where MU is the upper bound on the spacecraft's initial mass. Finally, time/epoch defects are multiplied by $1/TU$.

4.2 Body

Donald

Each `body` class owns a `frame` object and a `bplane` object.

4.3 CentralBody

4.4 Frame

The `Frame` class handles transformations between EMTG's available reference frames. EMTG supports the following reference frames:

- **`ICRF`**: A frame aligned with the International Celestial Reference Frame (ICRF) as defined in the IAU Cartographic Coordinates and Rotational Elements document [1] and in SPICE [2]. All of EMTG's internal calculations are performed in ICRF, so if the user specifies a state in some other frame, then the `frame` class has to transform it.
- **`J2000_BCI (J2000 body-centered inertial)`**: A frame defined by the first two Euler rotations (α and δ) in the IAU Cartographic Coordinates and Rotational Elements document [1], evaluated at January 1st, 2000 (J2000). This frame uses only the constant terms of the rotation, since the time-dependent terms are expressed as time since the J2000 epoch.
- **`J2000_BCF (J2000 body-centered fixed)`**: A frame defined by a third Euler rotation beyond `J2000_BCI` - a rotation about the z axis by W as defined in the IAU Cartographic Coordinates and Rotational Elements document [1]. The first two rotations are evaluated at the J2000 epoch, but the W rotation is evaluated at the epoch of the state being rotated. EMTGv9 only supports the linear component of the rotation, *i.e* $W = W_0 + \dot{W}$. This was judged sufficient for EMTG's intended use cases. EMTG's `J2000_BCF` is still an inertial frame; the time derivatives of the unit vectors of the `J2000_BCF` frame are all zero. In other words, the position and velocity are rotated according to the IAU angles, but the velocity transformation does not include the $\omega \times \mathbf{r}$ term. This is deliberate and is a definition that is often used for designing planetary probe trajectories. Kyle, Noble
- **`TrueOfDate_BCI (true-of-date body-centered inertial)`**: A frame identical to `J2000_BCI` except that all three Euler rotations are evaluated at the epoch of the state being rotated.

- **`TrueOfDate_BCF (true-of-date body-centered fixed)`**: A frame identical to `J2000_BCF` except that all three Euler rotations are evaluated at the epoch of the state being rotated.
- **`PrincipleAxes`**: Noble. Not yet fully implemented.
- **`Topocentric`**: Noble. Not yet fully implemented.
- **`Polar`**: Noble. Not yet fully implemented.
- **`J2000_BCR (J2000 body-centered rotating)`**: A frame identical to `J2000_BCF` except that the velocity vector is with respect to the rotating BCF frame. In other words, the $\omega \times \mathbf{r}$ term is included, unlike in the `J2000_BCF` frame. Not yet implemented.
- **`TrueOfDate_BCR (true-of-date body-centered rotating)`**: A frame identical to `J2000_BCR` except that all three Euler rotations are evaluated at the epoch of the state being rotated. Not yet implemented.
- **RIC (radial-intrack-crosstrack)**: A frame defined by the position and velocity vectors of the central body relative to some other reference body. For example, OSIRIS-REx uses a RIC frame where the `x` axis is the vector from the Sun to Bennu $\mathbf{r}$, the `z` axis is the angular momentum vector of Bennu about the sun $\mathbf{h} = \mathbf{r} \times \mathbf{v}$, and the `y` axis completes the right-handed set, $\mathbf{h} \times \mathbf{v}$. (not yet implemented)
- **SAM (sun-angular momentum, also called sun-north)**: The x axis points from the central body to a reference body (*i.e.* the Sun). The y axis is the cross product of the `J2000BCI` spin pole and the x axis. The z axis complete the right-handed set.
- **Object-Referenced**: The x axis points from the central body to the reference body. The y axis is the velocity vector of the reference body relative to the central body. The z axis completes the right-handed set and is the angular momentum vector of the reference body with respect to the central body. When this frame is used with a free point boundary condition, the first and second entries in the journey's destination list define the reference body for the departure and arrival event, respectively.

At construction, the owning object passes in the three Euler rotation angles and their first-order derivatives $(\alpha_0, \dot{\alpha}, \delta_0, \dot{\delta}, W, \dot{W})$. The `initialize_J2000()` method is called by the constructor and creates the rotation matrices between `ICRF` and `J2000_BCI`. These rotations are immediately available. The time-dependent transformations not evaluated immediately because the epoch of interest is not yet known. Instead, they are evaluated when the owning object, or another method of the `frame` itself, calls the `construct_rotation_matrices()` method. The rotations can be constructed with or without derivative information, as needed.

The owning object may then perform the transformation either by calling *get_R()* and `get_dRdt` and performing the transformation outside the frame class, or more safely, by calling the `rotate_frame_to_frame()` method. The `rotate_frame_to_frame()` method performs the transformation and returns both the transformed state and its derivatives.

The `rotate_frame_to_frame()` method will be the only safe way to transform to and from `Topocentric`, `Polar`, `J2000_BCR`, and `TrueOfDate_BCR` once they are fully implemented. We plan to force EMTG to use `rotate_frame_to_frame()` everywhere, just to make sure.

4.5 StateRepresentation

The `StateRepresentation` set of classes handle the transformation between state representations. EMTG's default state representation is Cartesian. However there are times in various parts of the program where the user might specify state in another representation. This most frequently happens in `ParallelShootingStep` and `FreePointBoundary` but could be used in other places as well.

`StateRepresentationBase` is an abstract base class for state representation classes. A new derived class is written for every new state representation. These classes handle the transformation from Cartesian to the new state representation and back, and also compute the partial derivatives. It is up to the host class to put these partial derivatives into the Jacobian - `StateRepresentationBase` and its children just puts the math all in one place.

EMTG currently implements the following state representations, each of which has its own derived class:

- Cartesian
- SphericalAZFPA
- SphericalRADEC
- Classical Orbit Elements (COE)
- B-plane coordinates $\{v_\infty, RHA, DHA, b_{radius}, b_{theta}, TA\}$. A full math spec is available in the "bplane_math.pdf" file in the EMTG repository. Note that the b-plane reference vector for `IncomingBplaneRepresentation` is the current reference frameś $\hat{z}$ vector. It is not possible to make the reference vector depend on the state as that would result in a circular definition of the Cartesian state as a function of the b-plane coordinates.

4.6 B-plane

Donald

4.7 AccelerationModel

The `AccelerationModel` abstract base class establishes the interface between any derived acceleration models and EMTG's equations of motion classes (4.9) as well as any other entities that may have a need to directly it (MGALT with perturbations). The methods `AccelerationModel::setEpoch` and `AccelerationModel::setEpochJD` are used to set the current epoch (in seconds past the J2000 epoch and the Julian date in days respectively) to be used by any time-dependent acceleration terms (e.g. body rotational states, ephemeris-based gravity calculations,

atmospheric drag). The `AccelerationModel::computeAcceleration` method takes a state vector and a boolean flag as its two inputs. The bool flag specifies whether the `AccelerationModel` should perform derivative computation. This method computes the instantaneous acceleration and stores the resultant `acceleration` vector. An overloaded for `computeAcceleration` can also be defined that accepts an input control vector should the `AccelerationModel` require a control term. The current acceleration vector can be extracted from the `AccelerationModel` using `AccelerationModel::getAcceleration`. The `AccelerationModel::populateInstrumentationFile` method must be implemented and should write acceleration data to the input `ofstream` for the input epoch (this method is also overloaded with a version that accepts a control vector as an input).

4.7.1 SpacecraftAccelerationModel

The `SpacecraftAccelerationModel` class implements a Cartesian acceleration model for spacecraft motion with respect to the center of a gravitating body. A `SpacecraftAccelerationModel` can model the perturbative effects of n third body gravity sources, spherical body solar radiation pressure, central body oblateness (J2) and gas drag assuming an exponential atmosphere model.

Figure 4.1: AccelerationModel inheritance diagram.

A `SpacecraftAccelerationModel` constructor requires pointers to a `MissionOptions` object, a `JourneyOptions` object, a `Universe` object, a vector of decision variable descriptions and a `Spacecraft` object. The input parameter `num_STM_size_in` specifies the row and column dimensions the state propagation matrix that the acceleration model will compute. The boolean parameter `needCentralBody` indicates whether or not central body gravity should be included in the total acceleration calculation. This is useful for using a `SpacecraftAccelerationModel` to compute perturbations only (i.e. for an MGALT phase that models perturbation impulses, and accounts for two-body motion with respect to the central gravitating body using a `KeplerPropagator`).

On construction, the method `SpacecraftAccelerationModel::constructorInitialize` sizes various internal containers, tests if the acceleration is being computed in a heliocentric frame and then constructs whatever `SpacecraftAccelerationModelTerms` were requested by the user (via the `JourneyOptions` object pointer). During the construction of the individual `SpacecraftAccelerationModelTerms` the `needCentralBody` flag to determine if a `CentralBodyGravityTerm` is required. Calls to the `SpacecraftAccelerationModel` are categorized as heliocentric/non-heliocentric for the purposes of computing the distance from the sun for solar electric power calculations (non-heliocentric trajectories require an extra vector addition step and an associated ephemeris look-up to compute the sun-spacecraft distance).

The primary public method of the `SpacecraftAccelerationModel` is `SpacecraftAccelerationModel::computeAcceleration`, which computes and stores the acceleration experienced by the spacecraft. The inputs to this method are the spacecraft state vector, two flags indicating whether or not partial derivatives of the acceleration model should be computed and a control vector $\mathbf{u}$ (if the control overload is being called). Prior to calling this method, it is important to set the current epoch using `AccelerationModel::setEpoch` or `AccelerationModel::setEpochJD`. The majority of the public methods in `SpacecraftAccelerationModel` are get methods for extracting acceleration information from the model. These include `SpacecraftAccelerationModel::getAccelerationVec` (returns a vector of the total acceleration experienced by the spacecraft in the central body ICRF frame), `SpacecraftAccelerationModel::getGravityAccelerationVec` (returns a vector of the total gravitational acceleration on the spacecraft in the central body ICRF frame), `SpacecraftAccelerationModel::getGravityAccelSources` (returns a vector of 3-tuples, where each tuple contains a body name, body mu, and the acceleration of the spacecraft due to the body), `SpacecraftAccelerationModel::getSRPAccelerationVec` (returns the acceleration vector due to solar radiation pressure), `SpacecraftAccelerationModel::getThrustAccelerationVec` (returns the acceleration vector due to the spacecraft's thruster), `SpacecraftAccelerationModel::getThrusterMaxMassFlowR` (returns the mass flow rate of the thruster), `SpacecraftAccelerationModel::getControlNorm` (returns the norm of the control parameters $u_x,\ u_y,\ u_z$), `SpacecraftAccelerationModel::getDutyCycle` (returns the duty cycle of the propulsion system), `SpacecraftAccelerationModel::getfx` (returns the state propagation matrix), `SpacecraftAccelerationModel::getSTMsize` (returns the row/column dimension of the state transition matrix), and `SpacecraftAccelerationModel::getCB2SC` (returns the distance from the central body to the spacecraft).

The `SpacecraftAccelerationModel::populateInstrumentationFile` method includes an overload with control and generates acceleration data for the current epoch and spacecraft state vector and writes it to the input `std::ofstream`. To do so, this method first writes the current JD epoch, the spacecraft state vector (including mass) and the total acceleration vector. Then, the method calls `SpacecraftAccelerationModelTerms::populateInstrumentationFile` for each term included in the acceleration calculation.

The private method `SpacecraftAccelerationModel::initializeAndComputeGeometry` zeros out the acceleration vector and the state propagation matrix (but sets the $\frac{\partial \mathbf{v}}{\partial \mathbf{v}}$ submatrix equal to $\mathbb{I}_{3\times 3}$) and calls `SpacecraftAccelerationModel::computeScCbSunTriangle`. This method computes the position/velocity of the spacecraft w.r.t. the sun as well as the following partial derivative information:

$$\frac{\partial \mathbf{r}_{\odot}}{\partial t} \tag{4.1}$$

$$\frac{\partial \mathbf{r}_{\odot}}{\partial \Delta t_{\mathrm{p}}} \tag{4.2}$$

,

where Δt_{p} is the current phase flight time. If the trajectory is heliocentric, then Eq. (4.1) and (4.2) are both zero.

4.8 SpacecraftAccelerationModelTerm

This is the abstract base class for the individual acceleration term classes in the `SpacecraftAccelerationModel`. Every `SpacecraftAccelerationModel` owns a `boost::ptr_vector` of `SpacecraftAccelerationModelTerm` objects that are individually evaluated and polled for their contribution to the total acceleration acting on the spacecraft. The `SpacecraftAccelerationModelTerm` requires that each of its child classes define the following methods: `computeAccelerationTerm` (and its partial derivative overload), which evaluates the Cartesian acceleration contribution of the term, and `populateInstrumentationFile`, which writes acceleration data to the specified acceleration output file. The public accessor `getTermAcceleration` returns the computed acceleration contribution of the term. Each `SpacecraftAccelerationTerm` owns a member variable `SpacecraftAccelerationTerm::term_acceleration`, which stores the acceleration contribution of the term and `SpacecraftAccelerationTerm::acceleration_model`, which is a pointer to the `SpacecraftAccelerationModel` that owns the term object. Additional `SpacecraftAccelerationTerms` may be added to the `SpacecraftAccelerationModel` by creating them in the latter's constructor and adding their evaluation to `SpacecraftAccelerationModel::computeAcceleration`. The derived child classes that are currently implemented are described in the following subsections.

4.8.1 GravityTerm

`GravityTerm` contains the equations of motion that account for the acceleration due to the gravity of a point mass `Body` object on the spacecraft as well as on the `CentralBody` (center of integration). The `GravityTerm` constructor takes a pointer to its parent `SpacecraftAccelerationModel` and a pointer to the `Body` object whose gravity the class is modeling as input parameters. The `computeAccelerationTerm` override calls the primary physics method `computePointMassGravityAcceleration` and then, if necessary, `computePointMassGravityDerivatives`.

The `computePointMassGravityAcceleration` method performs two main functions: 1) compute the position/velocity vectors between the `CentralBody`, the spacecraft and the gravitating point mass `Body` via a call to `computeScBodyCBtriangle` and 2) compute the acceleration on the spacecraft due to the gravitating `Body`. If the `GravityTerm Body` is the Sun, then the call to `computeScBodyCBtriangle` is skipped (to forgo an expensive ephemeris lookup call) as the `SpacecraftAccelerationBody::computeScCbSunTriangle` method has already been executed and has stored the position/velocity of the Sun w.r.t. the spacecraft and the `CentralBody`. Both the direct and indirect contributions of the gravity of the `Body` on the spacecraft are computed. The resultant point-mass gravity acceleration due to i `Body` objects is computed in `SpacecraftAccelerationModel::computeAcceleration`:

$$\ddot{\mathbf{r}} = -\sum_{\mathrm{i}} G\, m_{\mathrm{i}} \left(\frac{\mathbf{r} - \mathbf{r}_{\mathrm{i}}}{\|\mathbf{r} - \mathbf{r}_{\mathrm{i}}\|^3} + \frac{\mathbf{r}_{\mathrm{i}}}{r_{\mathrm{i}}^3} \right) \tag{4.3}$$

4.8.1.1 CentralBodyGravityTerm

The `CentralBodyGravityTerm` class is derived from `GravityTerm` and is used to compute the acceleration of the (massless) spacecraft due to the gravity of the `CentralBody` whose geometric center coincides with the center of integration:

$$\ddot{\mathbf{r}} = -\frac{G\left(\cancel{m}^{0} + m_{\text{cb}}\right)}{r^2}\frac{\mathbf{r}}{r} \tag{4.4}$$

The `CentralBodyGravityTerm` overrides the indirect gravitational effect of point mass gravity in `GravityTerm::computeFrameDragAcceleration` and sets that term equal to zero (since the center of integration is the center of the `CentralBody`, the `CenterBody` does not have a gravitational influence on itself, i.e. it has no trajectory w.r.t. the center of integration). The `GravityTerm::computePointMassGravityTimeDerivatives` method is similarly overrided and does nothing as the `CentralBody` gravity acceleration has no explicit time partials.

The `CentralBodyGravity` object also computes the acceleration on the spacecraft due to any `SphericalHarmonicTerms` that it owns in the `gravitational_harmonic_terms boost::ptr_vector` container. At this time, only the J2 harmonic gravity term has been implemented, so the aforementioned container contains, at most, one `SphericalHarmonicTerm`.

4.8.2 SphericalHarmonicTerm

The `SphericalHarmonicTerm` class was designed to accommodate a spherical harmonic gravity model of arbitrary degree and order. Currently, only an oblateness model (J2) has been implemented. The `SphericalHarmonicTerm` constructor takes a pointer to the parent `SpacecraftAccelerationModel`, a pointer to the `Body` whose harmonic gravity this class is modeling, and a pointer to its parent/owner `CentralBodyGravityTerm`.

The `computeAccelerationTerm` method first extracts the position of the spacecraft w.r.t. the `CentralBody` by polling the `CentralBody` pointer. Next, it performs the following three actions: 1) rotate the spacecraft position into the true of date BCF frame 2) compute the acceleration due to the `CentralBody` oblateness 3) add the oblateness acceleration to the parent `CentralBody term_acceleration` variable. The `computeAccelerationTerm(bool)` overload computes the acceleration due to `CentralBody` oblateness and its partial derivatives.

$$\ddot{\mathbf{r}}_{J_2} = \ddot{\mathbf{r}}_{J_{2_{ICRF}}} = M_{BCF}^{ICRF}\ddot{\mathbf{r}}_{J_{2_{BCF}}} \tag{4.5}$$

$$\ddot{\mathbf{r}}_{J_{2_{BCF}}} = \frac{3J_2\mu R^2}{2r_{\text{BCF}}^5}\begin{bmatrix} 1-5\frac{x_{\text{BCF}}z_{\text{BCF}}^2}{r_{\text{BCF}}^2} \\ 1-5\frac{y_{\text{BCF}}z_{\text{BCF}}^2}{r_{\text{BCF}}^2} \\ 3-5\frac{z_{\text{BCF}}^3}{r_{\text{BCF}}^2} \end{bmatrix} \tag{4.6}$$

4.8.3 SolarRadiationPressureTerm

The `SolarRadiationPressureTerm` models spherical/cannonball solar radiation pressure. This term's constructor takes a pointer to its parent `SpacecraftAccelerationModel` as its only input parameter uses that pointer to extract the following parameters from the `MissionOptions` object: the spacecraft coefficient of reflectivity (SRP scale factor) C_r, the wetted surface area A_s, the illumination percentage K, solar irradiance ϕ at 1 AU (in W/m^2), and the speed of light in a vacuum c. The constructor also computes and stores the SRP coefficient:

$$C_{\text{SRP}} = \frac{C_r A_s K \phi}{c} \tag{4.7}$$

The `computeAccelerationTerm` method calculates the instantaneous acceleration due to SRP, which is directed along the Sun-spacecraft line:

$$\ddot{\mathbf{r}} = C_{\text{SRP}} \frac{1}{m\ r_{s/\odot}^2} \frac{\mathbf{r}_{s/\odot}}{r_{s/\odot}} \tag{4.8}$$

4.8.4 ThrustTerm

The `ThrustTerm` constructor takes a parent `SpacecraftAccelerationModel` pointer as its sole input in order to grant the class access to the `Spacecraft` (5.3) object to compute power system parameters, and the electric propulsion system performance parameters.

The `computeAccelerationTerm` method performs to actions to acquire the information it needs to compute the thruster acceleration 1) it extracts the control vector $\mathbf{u}$ from its parent `SpacecraftAccelerationModel` 2) the method requests that the `Spacecraft` object compute the current power system state and the electric propulsion system performance. The maximum thrust $T_{\max}$ and maximum mass flow rate $\dot{m}_{\max}$ are then extracted from the `Spacecraft` object. The acceleration due to the thrusters ($T_{\max}$ accounts for the case when multiple thrusters are firing as well as any duty cycle that has been applied to the actual maximum thrust level) is then computed as follows:

$$\ddot{\mathbf{r}} = \frac{T_{\max}}{m}\mathbf{u} \tag{4.9}$$

Similarly, the mass flow rate is computed:

$$\dot{m} = -\|\mathbf{u}\| \ \dot{m}_{\max} \tag{4.10}$$

If the `computeAccelerationTerm(bool)` overload is called, the `Spacecraft` model also provides partial derivative information for the power and thrust levels.

4.8.5 AerodynamicDragTerm

4.9 Equations of Motion

All equations of motion classes are instances of the Integrand class, which is operated on by an EMTG IntegrationScheme class in order to numerically integrate a state vector.

Figure 4.2: Integrand inheritance diagram.

The `TimeDomainSpacecraftEOM` class encodes the first order Cartesian differential equations of motion using time as the independent variable and is derived from the `Integrand` class, which is called by implementations of the EMTG `IntegrationScheme` class. Specifically, an `IntegrationScheme` implementation calls the `TimeDomainSpacecraftEOM::evaluate` method in order to compute the current values of the differential equations (at a given epoch) as well as the orbit variational equations:

$$\dot{\mathbf{r}} = \mathbf{v} \tag{4.11}$$
$$\dot{\mathbf{v}} = \mathbf{a} \tag{4.12}$$
$$\dot{m} = -\|\mathbf{u}\|\dot{m}_{\max} \tag{4.13}$$
$$\dot{t} = 1 \tag{4.14}$$
$$\dot{f}_{\text{virt}} = \dot{m}_{\text{ACS}} \tag{4.15}$$
$$\dot{o}_{\text{virt}} = \|\mathbf{u}\|\dot{m}_{\max} \tag{4.16}$$
$$\dot{\mathbf{\Phi}} = \mathbf{A}\mathbf{\Phi}, \tag{4.17}$$

where $\dot{m}_{\max}$ is the maximum mass flow rate, $\mathbf{u} = [u_x \;\; u_y \;\; uz]^T; \|\mathbf{u}\| \leq 1$ is the maneuver control vector (12.2.4) $\mathbf{A} = \frac{\partial \dot{\mathbf{X}}}{\partial \mathbf{X}}$ is the state propagation matrix and $\mathbf{X} = [\mathbf{r} \;\; \mathbf{v} \;\; m \;\; t \;\; f_{\text{virt}} \;\; o_{\text{virt}}]^T$. Note that a differential equation is included for the current epoch t (Eq. 4.14), which is used to compute the current epoch if the independent variable of integration is not time (e.g. a Sundman propagator).

The `TimeDomainSpacecraftEOM::evaluate` method has two overloads, one for use with a control input, and one without (for ballistic trajectories). The `TimeDomainSpacecraftEOM::evaluate` method, first calls `TimeDomainSpacecraftEOM::computeAcceleration` (which has overloads for control/no control), which first sets the current epoch in the SpacecraftAccelerationModel and passes it pointers that it will use to extract time of flight and epoch partials from the SpacecraftAccelerationModel. Then, `TimeDomainSpacecraftEOM::computeAcceleration` calls `SpacecraftAccelerationModel::computeAcceleration`, which computes the instantaneous acceleration of the spacecraft and stores that vector. Next, `TimeDomainSpacecraftEOM::evaluate` calls `TimeDomainSpacecraftEOM::ballisticEOM`, the method that computes the first order differential equations of motion for ballistic flight. To do this, it extracts the current acceleration vector from the `SpacecraftAccelerationModel`. Despite the fact that it computes ballistic EOMs, this method also computes the differential equation for virtual chemical fuel if ACS usage is being tracked (section 10.3). If the control overloaded `TimeDomainSpacecraftEOM::evaluate` is called, then `TimeDomainSpacecraftEOM::propulsionEOM` is called next, which computes the mass flow rate first order differential equations. Both overloads of `TimeDomainSpacecraftEOM::evaluate` set time derivative of current epoch (state vector entry 7) to 1.0, i.e. current epoch is (trivially) integrated, and it's time rate of change is 1.0. Finally, `TimeDomainSpacecraftEOM::evaluate` computes the variational equations by extracting the state propagation matrix from the SpacecraftAccelerationModel and computing the first order matrix differential equation.

The `SundmanDomainSpacecraftEOM` inherits from the `TimeDomainSpacecraftEOM` and applies the multiplicative transform to the t-domain EOM that transforms them to τ-domain. It also computes the modified variational equations that account for this transformation:

$$\mathring{\mathbf{X}} = \frac{d\mathbf{X}}{d\tau} = \left(\frac{d\mathbf{X}}{dt}\right)\frac{dt}{d\tau} = \dot{\mathbf{X}}c_\gamma r^\gamma = \dot{\mathbf{X}}\eta \tag{4.18}$$
$$\mathring{\mathbf{\Phi}} = \frac{\partial \mathring{\mathbf{X}}}{\partial \mathbf{X}} = \left(\frac{\partial \dot{\mathbf{X}}}{\partial \mathbf{X}}\eta + \dot{\mathbf{X}}\frac{\partial \eta}{\partial \mathbf{X}}\right)\mathbf{\Phi} = \left(\mathbf{A}\eta + \dot{\mathbf{X}}\frac{\partial \eta}{\partial \mathbf{X}}\right)\mathbf{\Phi} \tag{4.19}$$

where $\eta = c_\gamma r^\gamma$ is the multiplicative Sundman transformation function, c_γ is a constant, which is

set equal to the Universe length unit (LU), and γ is the Sundman power constant, which is set to 1.0 in the current implementation. This means that the integration time steps correspond to a eccentric anomaly (to within a constant scale factor).

Chapter 5

Hardware Models

5.1 Overview

All calculations in EMTGv9 that model the performance of hardware, *i.e.* a launch vehicle or a spacecraft, are handled by EMTG's hardware classes. These classes are deliberately designed such that they can be used both by EMTG itself and by external programs. For example, the classes described in this chapter are also used by General Mission Analysis Toolkit (GMAT) and can also be compiled as a Python module.

The purpose of the separable hardware models is threefold. First, we wanted to create hardware models that could be used by many tools, not just EMTG. Second, we wanted a way to configuration manage spacecraft and launch vehicle properties just like we do ephemeris files. Third, we wanted all information about physical hardware to reside in text files that are not distributed with EMTG. That way we can distribute EMTG without any proprietary or ITAR/EAR restricted information.

5.2 Launch Vehicle Model

The `LaunchVehicle` class is responsible for calculating the performance (delivered mass) of a launch vehicle to an input hyperbolic excess energy (C_3) as well as the derivative of the delivered mass with respect to a changing C_3. EMTG currently supports only a polnomial function of mass as a function of C_3,

$$m\left(C_3\right) = \left(1 - \eta_{LV}\right) \sum_{i=0}^{n} c_i C_3^i - m_{adapter} \tag{5.1}$$

where η_{LV} is launch vehicle margin as specified in the user's .emtgopt mission script, the c_i are coefficients associated with the particular launch vehicle, and $m_{adapter}$ is the launch vehicle adapter mass specified.

The LaunchVehicle class is designed such that additional launch vehicle models may be added at a later date.

5.2.1 Launch Vehicle Library

The coefficients necessary to define a LaunchVehicle, as well as the upper and lower bounds on the validity of the performance polynomial and the upper and lower bounds on the Declination of Launch Asymptote (DLA), are held in a LaunchVehicleOptions object. A set of LaunchVehicleOptions objects are held in a LaunchVehicleOptionsLibrary.

LaunchVehicleOptionsLibrary reads a text file defining a set of launch vehicles. Each launch vehicle is associated with a "launch vehicle key string." The user provides a key string to EMTG, and EMTG in turn passes the key string into the LaunchVehicleOptionsLibrary and returns the particular LaunchVehicleOptions object that the user wants. The LaunchVehicleOptions object is then passed as an argument to the constructor of LaunchVehicle to create the calculation object. The LaunchVehicleOptionsFactory function performs these steps.

5.3 Spacecraft Model

The Spacecraft class handles all modeling of properties of the spacecraft propulsion and power system, as well as book-keeping of any propellant tank constraints. A spacecraft in EMTG is composed of a number of stages. The primary job of the spacecraft class is to hold a vector of Stage objects as described in Section 5.4. The Spacecraft class also tracks the indices of all of the virtual propellant tank variables in the optimization problem so that EMTG can perform global accounting of chemical fuel, chemical oxidizer, and electric propellant as described in Section 10.3. The rest of EMTG interacts with all properties of the Stage objects via the Spacecraft interface.

The Spacecraft object and its children are populated based on the contents of a SpacecraftOptions object. Further information about SpacecraftOptions is contained in Section **??**.

5.4 Stage Model

A Spacecraft contains one or more Stage objects. Each Stage object describes the power system, thrusters, and propellant tanks for one stage of a spacecraft. If the spacecraft's active Stage is changed, then the power, propulsion, and tank properties also change. This can be used for true multi-stage spacecraft or just to change propulsion and power characteristics to represent hardware degradation late in a mission.

Each Stage object contains a PowerSystem 5.5 object, a ChemicalPropulsionSystem 5.6.1, an ElectricPropulsionSystem 5.6.2, and vectors of decision variable indexes corresponding to stage-specific propellant tank constraints.

`Stage` objects are configured based on the contents of a `StageOptions` object as described in Section **??**.

5.5 Power System Model

The `PowerSystem` object computes the power properties of the spacecraft as a function of position in the solar system and time since launch. EMTG has two different models of solar power systems and one fixed power model. All three models can also be subjected to time degradation.

The first and simplest of EMTG's power models is a constant power,

$$P_{produced,0} = P_0 \tag{5.2}$$

where $P_{produced,0}$ is the output of the spacecraft's power supply at the beginning of the mission and P_0 is a quantity supplied by the user.

There are also two models for a solar power system. The first is a polynomial model,

$$P_{produced,0}(r) = \frac{P_0}{r^2}\left(\gamma_0 + \gamma_1 r + \gamma_2 r^2 + \gamma_3 r^3 + \gamma_4 r^4 + \gamma_5 r^5 + \gamma_6 r^6\right) \tag{5.3}$$

where the γ_i are user supplied coefficients fit to ground test data of a particular solar cell and r is the distance from the spacecraft to the sun in AU. The second solar power model, using the same coefficients, was developed by Carl Sauer for the SEPTOP tool,

$$P_{produced,0}(r) = \frac{P_0}{r^2}\left(\frac{\gamma_0 + \gamma_1/r + \gamma_2/r^2}{1 + \gamma_3 r + \gamma_4 r^2}\right) \tag{5.4}$$

The choice of appropriate solar power model for each application is based on which one better fits the cell performance data supplied by the manufacturer.

Time decay may then be applied to any of the three power supply models, yielding an expressing for the power produced by the spacecraft,

$$P_{produced}(r,t) = P_{produced,0}(r) \exp -\lambda t \tag{5.5}$$

where *lambda* is a user-supplied rate constant and t is the number of years since the user-defined power system reference epoch.

The power available to the propulsion system is then calculated as,

$$P_{available}(r,t) = (1 - \eta_{power}) * (P_{produced}(r,t) - P_{bus}(r)) \tag{5.6}$$

where η_{power} is propulsion power margin and is supplied in the user's .emtgopt input file, and $P_{bus}(r)$ is the power required by the spacecraft bus for non-propulsion functions.

EMTG has two different models to compute $P_{bus}(r)$. The first is a quadratic model,

$$P_{produced,0}(r) = \beta_0 + \beta_1/r + \beta_2/r^2 \tag{5.7}$$

where the β_i are user-supplied coefficients. There is also a conditional model,

$$P_{produced,0}(r) = \beta_0 \qquad \text{if } P_{produced}(r,t) > \beta_0 \tag{5.8}$$

$$= \beta_0 + \beta_1(\beta_2 - (P_{produced}(r,t)) \qquad \text{otherwise} \tag{5.9}$$

The user may select any combination of power supply and bus power models, with or without time decay. The time decay mode currently does not work properly with the Multiple Gravity Assist with Low-Thrust (MGALT) phase transcription 12.2.4.1 because there are some issues with chaining the time derivatives of the match point constraints.

5.6 Propulsion System Model

Each `Stage` object contains a `ChemicalPropulsionSystem` and an `ElectricPropulsionSystem`. Both inherit from the `PropulsionSystem` abstract base class. `PropulsionSystem` holds values for I_{sp}, thrust, and system mass. None of these need to be constants - the derived classes can compute them based on other supplied information. `PropulsionSystem` also defines interfaces common to both `ChemicalPropulsionSystem` and `ElectricPropulsionSystem`.

5.6.1 Chemical Propulsion System Model

EMTG's `ChemicalPropulsionSystem` class computes the propellant consumed by the spacecraft in both biprop and monoprop modes. Fuel and oxidizer consumption for a maneuver are computed along with their derivatives with respect to decision variables. The user provides the I_{sp} of the monoprop and biprop modes, along with a mixture ratio. EMTG assumes that all major maneuvers (maneuvers chosen by the optimizer) are biprop and all proportional TCMs and attitude control mass drops are monoprop. The user may individually override any biprop maneuver to make it monoprop. The `ChemicalPropulsionSystem` class may also be supplied with "mass per string" and "number of strings" quantities that are used to calculate the dry mass of the system.

EMTG currently models all chemical maneuvers as impulses. Thrust is only considered when writing a maneuver spec file. Only one thrust value is provided for the ChemicalPropulsionSystem, applicable to both monoprop and biprop modes. This is not technically correct. There should be two thrust values for each ChemicalPropulsionSystem object. This will be changed later.

5.6.2 Electric Propulsion System Model

The `ElectricPropulsionSystem` class computes the performance characteristics of the spacecraft's electric propulsion system. It can also size the system if provided with a "mass per string" and "number of strings." `ElectricPropulsionSystem` also computes the derivatives of thrust, I_{sp}, mass flow rate, *etc.* with respect to input power.

EMTG supports many types of electric thruster model, each of which is described in its own section below. The thruster models return the active thrust, I_{sp}, mass flow rate $\dot{m}$, and the "active power" P_{active} used by the propulsion system. P_{active} may be equal to or less than $P_{available}$. All thrust and mass flow rate outputs are scaled by a user defined operational duty cycle. The thrust is scaled by an additional user-defined "thrust scale factor" that is used to represent canted thrusters.

5.6.2.1 Constant Thrust and I_{sp} electric thruster model

The simplest electric thruster model is constant thrust and I_{sp}. This model does not require an input power and returns zero for all derivatives. The propulsion system is modeled as a single "super thruster."

5.6.2.2 Fixed Efficiency, Constant I_{sp} electric thruster model

EMTG can model the performance of a propulsion system with fixed I_{sp} and propulsion system efficiency η_{prop}. The propulsion system is modeled as a single "super thruster." The user provides bounds on P_{active}. If $P_{available}$ exceeds $P_{active,max}$ then it is clipped to $P_{active,max}$. If $P_{available}$ is less than $P_{active,min}$ then no thrusting occurs. The thrust is computed as,

$$T = \frac{2000\eta_{prop}}{I_{sp}g_0}P_{active} \tag{5.10}$$

where g_0 is the acceleration due to gravity at sea level on Earth, 9.80665 m/s^2.

Mass flow rate is given by,

$$\dot{m} = \frac{T}{I_{sp}g_0} \tag{5.11}$$

The partial derivatives of thrust and mass flow rate with respect to input power are,

$$\frac{\partial T}{\partial P} = \frac{2000\eta_{prop}}{I_{sp}g_0} \tag{5.12}$$

$$\frac{\partial \dot{m}}{\partial P} = \frac{1}{I_{sp}g_0}\frac{\partial T}{\partial P} \tag{5.13}$$

5.6.2.3 1D Polynomial electric thruster model

The next higher level of fidelity is a polynomial approximation of thrust and I_{sp} as a function of input power. Individual thrusters, and so the `ElectricPropulsionSystem::compute_number_of_active_thrusters` is called as described in Section 5.6.2.6. `ElectricPropulsionSystem::compute_number_of_active_thrusters` returns the number of active strings and the input power per string. The thrust and mass flow rate per string is then computed as,

$$T = a_t + b_t P + c_t P^2 + d_t P^3 + e_t P^4 + f_t P^5 + g_t P^6 \tag{5.14}$$

$$\dot{m} = a_f + b_f P + c_f P^2 + d_f P^3 + e_f P^4 + f_f P^5 + g_f P^6 \tag{5.15}$$

$$\tag{5.16}$$

where the coefficients are all supplied by the user. Equations 5.16 define a single string and are multiplied by the number of active strings to determine the total thrust and mass flow rate of the system. The input power per string is multiplied by the number of active strings to get the total P_{active}.

5.6.2.4 1D Smooth-stepped electric thruster model

While electric thruster performance is often modeled as a smooth function as in Section 5.6.2.3, in actuality they are operated at set performance points. It is typically possible to adjust mass flow rate and input voltage on grid. This results in discrete throttle settings. The propulsion system can take on any of these discrete settings if enough power is available.

EMTG's 1D smooth-stepped electric thruster model operates directly on the discrete throttle points. Full details of the motivation and calculations behind the 1D smooth-stepped model may be found in Knittel *et al.* [3] and are summarized here. The user supplies a list of throttle points that are read into EMTG `ThrottleSetting` objects and a sorting rule. The `ThrottleTable` class then creates non-dominated set of `ThrottleSetting` objects based on the sorting rule. The currently available sorting rules are:

- highest thrust per available power
- lowest mass flow rate per available power
- highest system efficiency per available power
- highest I_{sp} per available power
- full set - use all user-supplied throttle points

A gradient based optimization tool like EMTG requires thrust and I_{sp} to be first-differentiable with respect to input power and therefore the step function cannot be used directly. This is possible using the so-called Heaviside step function (see equation 5.17) [4]. A Heaviside function has a value of one for an input greater than or equal to a critical value, and zero for all other inputs. Each

throttle point is modeled with a Heaviside function that turns its value of thrust and mass flow rate "on" at its input thruster power. However, the magnitude of each Heaviside function only reflects a step increase over the previous throttle point's value. Let P_{a} be the continuously defined power available to the PPU, and $x_i \in \{x_1, x_2, \ldots, x_n\}$ denote either a mass flow rate or thrust magnitude set point associated with a power level $P_i \in \{P_1, P_2, \ldots, P_n\}$.

$$H_i = \begin{cases} 1 & \text{if } S_i \geq 0 \\ 0 & \text{if } S_i < 0 \end{cases} \tag{5.17}$$

$$S_i = P_{\mathrm{a}} - P_i \tag{5.18}$$

The mass flow rate or thrust magnitude available at a given power available is then:

$$x = x_1 H_1 + \sum_{i=2}^{n} (x_i - x_{i-1}) H_i \tag{5.19}$$

Implementing equation 5.19 solves the first of the three criteria set out above, in that the model output perfectly matches the capability of the thruster at any input power. However, this model does not meet the second criteria. The differentiability problem is solved by replacing the discontinuous Heaviside functions with logistics functions (Figure **??**) which smoothly and continuously approximate step increases:

$$\bar{H}_i = \frac{1}{1 + e^{-kS_i}} \tag{5.20}$$

The parameter k, or throttle sharpness, determines how closely the logistics functions model the step increases of the Heaviside functions.

The net model which solves the first two criteria set out above we name the 1-D stepped model and is presented in equation 5.21:

$$x = x_1 \bar{H}_1 + \sum_{i=2}^{n} (x_i - x_{i-1}) \bar{H}_i \tag{5.21}$$

and the relevant analytical derivative is:

$$\frac{dx}{dP_{\mathrm{a}}} = k x_1 \bar{H}_1^2 e^{-kS_i} + k \sum_{i=2}^{n} (x_i - x_{i-1}) \bar{H}_i^2 e^{-kS_i} \tag{5.22}$$

For a conceptual example of how this model works, let throttle levels 1 and 2 be consecutive members of a chosen non-dominated set requiring 1 kW and 2 kW to operate and generating 100 mN and 150 mN of thrust, respectively. As the power available to the thrusters increases past 2 kW, the Heaviside function for throttle level 2 switches "on." However, because the engine is already outputting 100 mN from throttle point 1, the step increase associated with throttle level 2's Heaviside function is only 50 mN resulting in a net output of 150 mN. The input power has not

yet been reached for any of the other throttle levels, so they do not contribute to the "net" thrust and mass flow rate, even though they are summed over.

For values of k greater than 1000, the Heaviside step function and the logistics function approximation are nearly indistinguishable. However, this has negative effects for the differentiability of the function. As k increases, the derivatives with respect to power in the "flat" regions grow closer and closer to zero, and the derivative for $P_{\mathrm{a}} = P_i$ becomes larger and larger. In practice, the value of k should be chosen for the specific throttle box, and the capability of the gradient-based optimizer. Further, there could be a separate value of k chosen for each throttle level, if so desired, but here we assume that one identical value is used everywhere.

The user provides the throttle points in the form of a throttle table file. In practice, the optimizer performs poorly when multiplying two logistics function smoothers together. This is probably because the derivatives of the logistics function are very sharp. It is therefore a good idea to encode multi-thruster throttle points directly into the throttle table file instead of having the file represent a single thruster and instructing EMTG to also switch thrusters on and off.

5.6.2.5 2D Throttling

While the 1D polynomial and smooth-stepped models presented earlier are adequate in most circumstances, they still require the user to choose *a priori* how to traverse the throttle grid. It is preferable to make the entire range of throttle settings available and let the optimizer choose. This is an in-progress capability in EMTG. The spacecraft model supports it but the phase transcriptions currently do not.

EMTG's hardware classes support two 2D thruster models. One is a fixed efficiency, variable specific impulse approximation and one is a 2D smooth-step. Both compute thrust and mass flow rate as a function of input power and a control parameter $u_{command}$. At the time of this writing, neither of these 2D modes is supported by any of the phase transcriptions.

5.6.2.5.1 Fixed efficiency, Variable Specific Impulse electric thruster model The propulsion system may be modeled with a fixed efficiency and a variable I_{sp}. This is identical to the model described in Section 5.6.2.2 except that the I_{sp} is a control variable. In EMTGv8, the low-thrust transcriptions could be configured to allow the optimizer to choose a $u_{command}$, representing I_{sp}, at each control step. This is not yet implemented in EMTGv9. However the modeling does exist and is described here.

The I_{sp} is given by,

$$I_{sp} = (I_{sp,max} - I_{sp,min})\, u_{command} + I_{sp,min} \tag{5.23}$$

As in 5.6.2.2, the thrust and mass flow rate, and their derivatives with respect to power, are governed by Equations 5.10-5.13. The only difference is that there now exist partial derivatives of thrust and mass flow rate with respect to $u_{command}$.

$$\frac{\partial T}{\partial u_{command}} = -\frac{T}{I_{sp}} \left(I_{sp,max} - I_{sp,min} \right) \tag{5.24}$$

$$\frac{\partial \dot{m}}{\partial u_{command}} = -\frac{2\dot{m}}{I_{sp}} \left(I_{sp,max} - I_{sp,min} \right) \tag{5.25}$$

5.6.2.5.2 2D Smooth-Stepped electric thruster model The `ElectricPropulsionSystem` class also contains a 2D smooth-stepped model that allows the optimizer to traverse the entire throttle grid. Full details of this model may be found in Reference [3]. As in Section 5.6.2.5.1, an additional control parameter $u_{command}$ is necessary. Any parameter which has a discrete value at each throttle setting could be used (thrust, I_{sp}, voltage, current, $\dot{m}$), but the fewer settings that the control variable has, the more effective it will be. This model is supported by the ElectricPropulsionSystem class but is not supported by the phase transcriptions.

Similar to the 1-D model, the 2-D model uses the logistics function approximation to the Heaviside step function to turn each throttle point "on", however in the 2-D model, each throttle point uses Heaviside functions to turn the throttle level "off" as well. Because of the increased dimensionality, there is no *a priori* order to the throttle points. From a given pair of P_{a} and u_{command}, depending which (or both) control variable changes to move in the throttle grid, many different throttle levels could be the next to be activated. This creates a 2-D region over which each throttle point is active.

As the throttle table is initially parsed, the values of $P_{\text{on},i}$, $P_{\text{off},i}$, $\dot{m}_{\text{on},i}$, and $\dot{m}_{\text{off},i}$ are found for each throttle level. The "on" value for each throttle level always corresponds to the operating conditions of the given throttle level. Power and the command variable are sorted in slightly different manners to determine the "off" condition. All of the throttle points are sorted by their required voltage input. For a given throttle level, the critical value of voltage which turns it "off" by way of a negative Heaviside step function is the next greater voltage level, regardless of required input power. Unless, for the given power level, there are no additional valid throttle points at a higher voltage level, in which case no value of u_{voltage} would turn this point "off". On the other hand, the critical available power is found by sorting only those throttle levels with identical input voltage and selecting the next greater input power. If no such point exists, then there is no available power which delivers an "off" command for this throttle point.

Given both sorting methods, the 2-D model will be most effective by selecting a command variable with few discrete levels. For example, even though the NASA's Evolutionary Xenon Thruster (NEXT) thruster has 40 total throttle settings, there are only 12 distinct values of voltage and 8 distinct values of mass flow rate [5]. If there are many distinct command variable settings, then there will be more switching that occurs as power available varies. The more switches needed, the greater the computation time for the model. Further, if two distinct command variable settings are very close to each other, the value of k is constrained lest two settings could be transitioning "on" at the same input power.

As with the 1-D model, the net thrust output by the EP system is a summation of the thrust and mass flow rate functions for all throttle points. Assuming that the command variable is voltage,

the thrust and mass flow rate generated is:

$$x = \sum_{i=1}^{n} x_i (\bar{H}_{P_{\text{on},i}} - \bar{H}_{P_{\text{off},i}})(\bar{H}_{V_{\text{on},i}} - \bar{H}_{V_{\text{off},i}}) \tag{5.26}$$

Recalling that $\bar{H}_P$ was defined in equation 5.20, we simply now add a more detailed subscript to distinguish between the command variable and power available. $\bar{H}_V$ takes the same form, simply with critical values of voltage instead of power:

$$\bar{H}_{V_i} = \frac{1}{1 + e^{-k(u_{\text{voltage}} - V_i)}} \tag{5.27}$$

The derivatives with respect to decision variables are defined in the following equations (again with voltage as the command variable):

$$\frac{\partial x}{\partial P_{\text{a}}} = k \sum_{i=1}^{n} x_i (\bar{H}_{V_{\text{on},i}} - \bar{H}_{V_{\text{off},i}})(e^{-k(P_{\text{a}} - P_{\text{on},i})} \bar{H}^2_{P_{\text{on},i}} - e^{-k(P_{\text{a}} - P_{\text{off},i})} \bar{H}^2_{P_{\text{off},i}}) \tag{5.28}$$

$$\frac{\partial x}{\partial u_{\text{voltage}}} = k \sum_{i=1}^{n} x_i (\bar{H}_{P_{\text{on},i}} - \bar{H}_{\text{off},i})(e^{-k(u_{\text{voltage}} - V_{\text{on},i})} \bar{H}^2_{V_{\text{on},i}} - e^{-k(u_{\text{voltage}} - V_{\text{off},i})} \bar{H}^2_{V_{\text{off},i}}) \tag{5.29}$$

5.6.2.6 Multi-thruster switching

`ElectricPropulsionSystem`'s `compute_number_of_active_thrusters()` method can determine how many of a set of thrusters to fire at a time, depending on $P_{available}$ as computed in Equation 5.6. The user defines how many thruster/power processing unit (PPU) strings are available on the spacecraft and `compute_number_of_active_thrusters()` determines how many should be operated and at what input power. EMTG assumes that all thruster/PPU strings are identical. Each string has a minimum power P_{min} and a maximum power P_{max}.

The user chooses between two throttle logic rules. `MaxThrusters` instructs `compute_number_of_active_thrusters()` to fire as many thrusters as possible for a given $P_{available}$. This usually results in thrusters being fired close to their P_{min}, resulting in poor performance. `MinThrusters` fires as few thrusters as possible to use the entire $P_{available}$, often resulting in thrusters being fired closer to their P_{max} and resulting in better performance. Both throttle logics are provided because the "best" choice is sometimes non-intuitive.

A discrete change in the number of operating thruster/PPU strings would result in a discontinuity in the optimization problem and confuse a gradient-based optimizer. Therefore `compute_number_of_active_thrusters()` uses a smoothed Heaviside step step function approach just like the smooth-stepped thruster model described in Section 5.6.2.4 [4].

In the context of multi-thruster switching, we define a set of Heaviside step functions $H_i(P)$,

$$H_i(P) = \frac{1}{1 + \exp(-2k(P - P_i^*))} \tag{5.30}$$

where each Heaviside step function $H_i(P)$ defines the switch state of the ith thruster and k defines the sharpness of the transition. The larger the value of k, the closer $H_i(P)$ approximates the Heaviside step function. However, while the derivatives $H_i'(P)$ increase as k increases, they remain finite and $H_i(P)$ remains continuous. We find that the optimizer behaves best when the derivatives are reasonably small (*i.e.* small k) but we also find that if k is too small then the thruster model

will frequently report a fractional non-integer $H_i(P)$. N_{active} may then be defined as,

$$N_{active} = \sum_{i=1}^{N} H_i(P) \tag{5.31}$$

To fully define N_{active}, it is necessary to define the transition powers P_i at which a thruster would be switched on and off. If the `MinThrusters` logic is used, then each P_i is an integer multiple of P_{max} except for P_1, which is equal to P_{min}. Alternatively if the `MaxThrusters` logic is used, then each P_i would then be an integer multiple of P_{min}. It is also possible to define other switching laws such as maximum thrust or maximum I_{sp}, in which case one would need to compute the P_i where those merit functions change as a function of number of thrusters.

As a practical note, we find that multiplying Heaviside smooth-step functions together causes difficulty for a gradient-based optimizer. We therefore recommend that the multi-thruster switching model and the smooth-stepped thruster model not be used together. Instead, expand the throttle table used by the smooth-stepped thruster model such that it includes multi-thruster operating points.

5.6.3 Configuring the Spacecraft Class

The `Spacecraft` class is configured by passing a `SpacecraftOptions` object into the `Spacecraft` constructor. There are three ways to configure a `SpacecraftOptions` object - via programmatic assignment, by choosing systems out of library files, or by supplying a spacecraft file. All three are available in EMTG via the `SpacecraftOptionsFactory()` function and are also available when the hardware classes are used by other programs or accessed directly from Python. We encourage anyone who wishes to use EMTG's hardware model classes in their own code to start from the examples in `SpacecraftOptionsFactory()`.

5.6.3.1 Constructing the Spacecraft by Programmatic Assignment

It is possible to construct a `SpacecraftOptions` object entirely via `set()` methods. EMTG has a mode that does this based on choices made by the user in the .emtgopt script. This mode exists because we wanted something as similar as possible to how **EMTGv8!** (**EMTGv8!**) worked. Based on user inputs, `SpacecraftOptionsFactory` creates `PowerSystemOptions`, `ElectricPropulsionSystemOptions`, and `ChemicalPropulsionSystemOptions` objects. It creates a single `StageOptions` object and adds the `PowerSystemOptions`, `ElectricPropulsionSystem`, and `ChemicalPropulsionSystem` to it. Finally it creates a `SpacecraftOptions` object with a single stage. Global propellant tank and dry mass constraints are enabled if appropriate.

The `get_thruster_coefficients_from_library()` function contains a small hard-coded library of public-domain thruster polynomial coefficients, sourced from published papers. This is the only hardware data anywhere in the EMTG code base. If operating in legacy mode, the the `ElectricPropulsionSystem` object will be configured with one of these coefficient sets.

Programmatic construction of a `SpacecraftOptions` object is a flexible way for developers to

use EMTG's hardware models with their own program's user interfaces.

5.6.3.2 Constructing the Spacecraft from Library Files

A `SpacecraftOptions` object may also be constructed by choosing components from hardware library files. In this mode, the user specifies paths to a .emtg_powersystemsopt and a .emtg_-propulsionsystemopt file. The user also specifies key strings for the power system, the chemical propulsion system, and the electric propulsion system. The files are then parsed and the `Power-SystemOptions`, `ChemicalPropulsionSystemOptions`, and `ElectricPropulsionSystemOptions` objects are created. These are then attached to a `StageOptions` object, which in turn is attached to a `SpacecraftOptions` object. Global propellant tank and dry mass constraints are enabled if appropriate.

The method of constructing `SpacecraftOptions` from libraries allows the user to keep a database of different components and very quickly set up trade studies that explore the space of trajectory and systems options. We recommend that the library files be configuration managed.

5.6.3.3 Constructing the Spacecraft from a Spacecraft File

Finally, a `SpacecraftOptions` object may be constructed from a .emtg_spacecraftopt file. This is the most flexible of the spacecraft construction interfaces but also the most complex. A .emtg_-spacecraftopt file defines a spacecraft with one or more stages. Each stage block is self-contained and has libraries of power and propulsion systems, as well as key strings to tell `SpacecraftOptions` which system to use for which purpose. Each stage may have its own propellant tank and dry mass constraints in addition to the global constraints. The .emtg_spacecraftopt file is the only way to specify a multi-stage spacecraft to EMTG.

Regardless of which construction method is used, EMTG writes out a .emtg_spacecraftopt file. This file may then be re-used as input to future EMTG runs. We highly recommend that as soon as a spacecraft design stabilizes, it should be configuration managed.

Chapter 6

Propagation

Donald

Chapter 7

Integration

Numerical integration in EMTG is accomplished using a family of three associated classes: `IntegrationScheme`, `IntegrationCoefficients`, and `Integrand`. Broadly speaking, an `IntegrationScheme` uses an `IntegrationCoefficients` helper class to perform numerical integration of an `Integrand` (typically a spacecraft equations of motion class 4.9).

Figure 7.1: EMTG integration class inheritance diagram.

7.1 Integrand

The abstract base `Integrand` class defines an object that can be numerically integrated by an `IntegrationScheme` (see 7.2), i.e. it is used to implement a set of differential equations of motion. The `evaluate` method takes three inputs: 1) a state vector, 2) an input/output state rate-of-change vector, and 3) a bool flag that specifies whether partial derivatives of the equations of motion should

be computed. The `evaluate` method can also be overloaded to accept a control term input.

The `setCurrentIndependentVariable` method sets the current value of the independent variable of integration. For example, the `ExplicitRungeKutta` class calls this method in `ExplicitRungeKutta::evaluateIntegrand` immediately prior to calling `Integrand::evaluate` to ensure that the `current_independent_variable` member variable of `Integrand` has been updated should it need it.

The `Integrand` class must also populate a `state_propagation_matrix` container, containing the first order partial derivatives of the state differential equations of motion w.r.t. state vector (Jacobian matrix), which can then be retrieved using `Integrand::getStatePropMat`.

7.2 IntegrationScheme

`IntegrationScheme` is an abstract base class for the numerical integrators in EMTG. The `IntegrationScheme` class has two constructors. Both require a pointer to an `Integrand` object that it will integrate, and the one overload also allows the number of integration states `num_states` and the state transition matrix dimension `STM_size` to be specified on construction.

The primary purpose of this class is to require all of its child classes to define a `step` method, which integrates the `Integrand` equations by one step. It also provides the framework for a derived class to define an `errorControlledStep` for adaptive-step schemes and to compute an associated error via the `computeError` method.

Prior to calling the `step` method, the `setLeftHandIndependentVariablePtr` method must be called to setup the left-hand independent variable of integration (e.g. time, or an orbital anomaly).

The `step` method takes the following inputs: the left hand state vector and STM matrix (`state_left` and `STM_left`), output containers for the final integrated state and STM (`state_right` and `STM_right`), the integration step size (`step_size`), the partial derivative of the step size w.r.t. the phase propagation length decision variable (`dstep_dProp_var`) and a boolean variable (`needSTM`) that determines whether or not the STM variational equations should be integrated (and that is also passed along to the `Integrand` to inform whether or not it should compute partial derivatives). There is an overload for `step` that allows for a `control` vector to be specified.

An `errorControlledStep` method interface is also provided, which allows for the implementation of a `step` algorithm that also computes a state/STM error (via `computeError`) between two integrator solutions of different orders for the purposes of informing an adaptive-step propagator.

7.2.1 ExplicitRungeKutta

This class implements the explicit Runge-Kutta algorithm for iteratively solving a system of ordinary differential equations. The class is compatible for use with an explicit `RungeKuttaTableau` of arbitrary order (embedded or otherwise), and computes the first-order variational equations of the system equations of motion.

`ExplicitRungeKutta` defines two `IntegrationScheme::step` method overrides: one that includes a control vector input, and one without. The `step` overrides carry out the Runge-Kutta algorithm by calling one of four overloads of `stageLoop` (with/without control, with/without state transition matrix computations). The `stageLoop` methods take the left-hand state $\hat{\mathbf{X}}_n$ and STM $\hat{\phi}_n$ as inputs and compute the state at the right-hand side of the Runge-Kutta step $\hat{\mathbf{X}}_{n+1}$ as well as the right-hand STM $\hat{\phi}_{n+1}$ using the `stateUpdate` and `stmUpdate` methods respectively (the STM is only computed and updated if the appropriate `needSTM` flag is passed to the `step` method). To do this, it computes the first $s-1$ stages using the Runge-Kutta matrix coefficients a_{ij} that it extracts from the class-owned `RungeKuttaTableau`:

$$\mathbf{X}_i = \mathbf{X}_n + \sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j \qquad i > j \quad i,j = 1,2,3,...,s \tag{7.1}$$

$$\Phi_i = \Phi_n + \sum_{j=1}^{i-1} a_{ij}P_j \qquad i > j \quad i,j = 1,2,3,...,s \tag{7.2}$$

The specific type of `RungeKuttaTableau` used is determined at construction time. At each stage, the `Integrand` **f** is evaluated and stored in the `gradient_bin` container. Entries in this container are then multiplied by the integration step size in order to compute the stage sub-steps $\mathbf{k}_i$:

$$\mathbf{k}_i = h \cdot \mathbf{f}\left(t_n + c_i h,\ \mathbf{X}_i\right) \tag{7.3}$$

The matrices P_i are computed as follows, and are stored in the `STM_bin` vector of matrices:

$$P_i = \left(\left.\frac{\partial \mathbf{f}}{\partial \mathbf{X}}\right|_{\mathbf{X}_i} h + \mathbf{f}\left.\frac{\partial h}{\partial \mathbf{X}}\right|_{\mathbf{X}_i} \right) \Phi_i \tag{7.4}$$

In Eq. 7.4, $\frac{\partial \mathbf{f}}{\partial \mathbf{X}}$ is the state propagation matrix that is computed by the `Integrand` and is retrieved from that object using `Integrand::getStatePropMat`.

The Runge-Kutta nodes c_i are used to advance the independent variable of integration, which is represented here as time t, but can be any independent variable (e.g. an anomaly if Sundman-transformed equations of motion are being integrated). The final right-hand state, STM, and independent variable are then computed using the Runge-Kutta weights $\hat{b}_i$

$$\hat{\mathbf{X}}_{n+1} = \mathbf{X}_n + \sum_{i=1}^{s} \hat{b}_i \mathbf{k}_i, \qquad i = 2, 3, ..., s \tag{7.5}$$

$$\hat{\Phi}_{n+1} = \Phi_n + \sum_{i=1}^{s} \hat{b}_i P_i, \qquad i = 2, 3, ..., s \tag{7.6}$$

$$t_{n+1} = t_n + h \tag{7.7}$$

If the `ExplicitRungeKutta::errorControlledStep` method is called, then the embedded (lower) order right-hand solution is also computed using the lower order Runge-Kutta weights:

$$\mathbf{X}_{n+1} = \mathbf{X}_n + \sum_{i=1}^{s} b_i \mathbf{k}_i, \qquad i = 2, 3, ..., s \tag{7.8}$$

$$\Phi_{n+1} = \Phi_n + \sum_{i=1}^{s} b_i P_i, \qquad i = 2, 3, ..., s \tag{7.9}$$

For an embedded Runge-Kutta routine, the higher order state $\hat{\mathbf{X}}_{n+1}$ and STM $\hat{\Phi}_{n+1}$ are adopted as the solutions on the right-hand side of the Runge-Kutta step (`state_right` and `STM_right` respectively).

7.2.2 IntegrationSchemeFactory

7.3 IntegrationCoefficients

7.3.1 RungeKuttaTableau

The `RungeKuttaTableau` is a child class of `IntegrationCoefficients` and defines the Runge-Kutta matrix A, weights $\mathbf{b}$, and nodes $\mathbf{c}$ of a particular set of Runge-Kutta formulae.

7.3.1.1 RungeKuttaDP87Tableau

7.3.1.2 RungeKutta4Tableau

Chapter 8

SplineEphem

EMTG uses SPICE [2] its ephemeris source for solar system bodies. However SPICE requires repeated hard drive access and does not provide analytical partial derivatives of the state with respect to time. EMTG therefore does not use SPICE directly, but rather polls SPICE once at program bootstrap to build a table of state information for each body needed in a given EMTG run and then fits a spline to it. This technique offers a speed improvement of up to 80x over regular SPICE, and provides analytical derivatives, but uses more system memory. The user can choose the number of data points drawn from SPICE per period of the body and therefore can control the accuracy of the spline as a function of bootstrap time and memory footprint.

The `SplineEphem` library consists of the `SplineEphem_universe` and `SplineEphem_body`. `SplineEphem_universe` creates and manages a container of `SplineEphem_body` objects. `SplineEphem_body` fits splines to and interpolates as needed to find the position and velocity vectors, as well as their derivatives with respect to time, for a body relative to a user-defined reference body.

`SplineEphem_body` is built on top of the Gnu Scientific Library (GSL)'s clamped cubic spline functions as they are readily available and well tested. GSL is not compatible with EMTG's license and therefore we cannot distribute with GSL. Users must download their own GSL and link it to EMTG. At some later date the GSL spline library should be replaced with an unclamped spline library that can be distributed with EMTG. This is not in the critical path for anything but would be helpful. A side effect of using clamped splines is that the fit becomes less accurate near the bounds of the time interval. EMTG therefore fits a nine days before and after the requested time interval, just in case. Any new spline library should be fully compatible with algorithmic differentiation because the current method of extracting derivative information from GSL and then assigning it to derivative entries of operator-overloaded calculation objects is really, really annoying.

If for some reason the user does not wish to use `SplineEphem`, EMTG can use SPICE directly. Finite differencing is used to approximate the derivatives of the ephemeris with respect to time. This is slow and less robust than `SplineEphem` but is slightly more correct and has a much smaller memory footprint.

Chapter 9

Utilities

9.1 Writey_Thing

`Writey_thing` is a base class that handles most of EMTG's file writing features. `Journey`, `Phase`, and `BoundaryEventBase` all inherit from `writey_thing`, as do their various owned member classes. `writey_thing` implements the following methods:

- `write_output_line` - writes a line in the .emtg output file, subject to arguments from the caller. See Doxygen for details.
- `write_ephemeris_line` - writes a line in the .emtg_ephemeris file. `write_ephemeris_line` may be called with or without control and propulsion arguments, depending on the use case.
- `write_ephemeris_state` - writes the state (epoch, position, velocity and optional mass) component of an ephemeris line. Called only by `write_ephemeris_line`.

9.2 Sparsey_Thing

`Sparsey_thing` is a base class that stores the handles the construction of the problem Jacobian sparsity problem. `Sparsey_thing` also stores pointers to the vectors that define the NLP problem, *i.e.* `Xlowerbounds, Xupperbounds, Xdescriptions, X_scale_factors, F, Flowerbounds, Fupperbounds, Fdescriptions, A, Adescriptions, iAfun, jAvar, Gdescriptions, iGfun, and jGvar`. `Sparsey_thing` also keeps track of the `prefix` that precedes the text description of any decision variables or constraints owned by the derived class.

`Sparsey_thing` also implements a number of helper methods that aid in constructing the Jacobian sparsity pattern. Each of these methods, as described below, alias to identically named functions in the `solver_utilites` namespace. The purpose of these aliases is to simplify the call syntax so that the programmer does not have to pass in pointers to the NLP problem definition

vectors every time a new piece of code is written. `Sparsey_thing` both owns those pointers and handles passing them into the various `solver_utilities` functions.

- `create_sparsity_entry` - Create a single entry in the Jacobian sparsity pattern and update `Gdescriptions` appropriately. `create_sparsity_entry` may be passed either a scalar helper index or a vector of helper indices. If passed a vector, `create_sparsity_entry` will append it. `create_sparsity_entry` must be given the Findex of the constraint of interest and either the Xindex of the decision variable that it has a derivative with respect to or a search string, starting location, and search direction in the Xdescriptions vector.
- `create_sparsity_vector` - Create several entries in the Jacobian sparsity pattern. This works the same way as `create_sparsity_entry` except that the caller `must` pass in a vector of helper indices, a search string, a starting location in Xdescriptions, a search direction, and a number of entries to add. `create_sparsity_vector` will find the entry corresponding to the search string and then add not only that decision variable but also the next caller-supplied n variables to the sparsity pattern.

9.3 Solver Utilities

`Solver_utilities` is a namespace of helper functions that construct the Jacobian sparsity pattern. They are fully described in Section 9.2 and are aliased by methods of `sparsey_thing`. In addition, `solver_utilities` includes the `detect_duplicate_Jacobian_entries()` helper function, which does exactly what it sounds like.

9.4 String Utilities

`String_utilities` is a namespace for string manipulation functions. Right now it contains only one, `convert_number_to_formatted_string`, which does exactly what is sounds like and is third-party code sourced from a Stack Overflow post at `http://stackoverflow.com/questions/7132957/c-scientific-notation-format-number`.

9.5 File Utilities

`File_utilities` is a namespace that contains file manipulation functions:

- `safeGetLine()` - A "safe" version of `std::getline()` that is agnostic to line termination character. This is very important for cross-platform compatibility.
- `get_all_files_with_extension` uses `boost::filesytem` to make a list of all files in a directory with a given extension. This is useful, for example, to find all of the .bsp files in one's `Universe/ephemeris_files` directory.

9.6 Maneuver spec writer

The maneuver spec writer is a set of helper classes that create lines in the maneuver spec file that can be read by MONSTER, PIRATE, PyGMATscripter, *etc.*. There are two classes:

- `maneuver_spec_line` - A container class of `maneuver_spec_item` objects. A given maneuver may have multiple items in it if the thrust, mass flow rate, *etc.* chane during the maneuver.
- `maneuver_spec_item` - A container of all of the data items that fully define a maneuver item, including thrust, mass flow rate, start epoch, duration, direction, duty cycle, initial and final mass, and Δv. `maneuver_spec_item` does the actual writing.

9.7 Target spec writer

`Target_spec_line` is a helper class for writing lines in a target spec file that can be read by MONSTER, PIRATE, PyGMATscripter, *etc.*. It stores the epoch, frame, central body, and state vector (position, velocity, mass) associated with the target. When appropriate, `target_spec_line` will also store and write BdotR, BdotT, and an ET seconds since J2000 representation of epoch.

9.8 MJD to Gregorian Date Converter

`Mjd_to_mdyhms`, supplied to the EMTG team by Brent Barbee, is a helper function that converts Modified Julian Date (MJD) to Gregorian month, day, year, hours, minutes, seconds. `Mjd_to_mdyhms` is used only in outputting .emtg summary files. We may eventually switch to SPICE for this, as we do in all other types of outputs.

9.9 Ephemeris Reader

`Ephemeris_reader` is a helper class to ingest and fit a spline to a .emtg_ephemeris file. This can be useful if one wants to track a reference trajectory. `Ephemeris_reader` uses the Gnu Scientific Library spline package.

9.10 Inverse Covariance Reader

`Covariance_reader` is a helper class that ingests and fits splines to a time-history of inverse covariance matrices. `Covariance_reader` uses the Gnu Scientific Library spline package.

Chapter 10

Mission

10.1 Overview

The `Mission` class derives from the core `problem` class and is a top-level container for the entire transcription of an EMTG mission. A mission contains a `boost::ptr_vector` of `Journey` objects, each representing one of the mission's journeys. The `Mission` also holds a `ObjectiveFunction` object, a vector of `Universe` objects, and `Spacecraft` and `LaunchVehicle` objects. `Mission` is responsible for the mission flight time constraint as described in Section 10.2, and any propellant or dry mass constraints as described in Section 10.3.

Each mission contains a vector of `Journey` objects, each of which contains in turn one or more `Phase` objects. Each `Phase` contains a `DepartureEvent` and an `ArrivalEvent`. This architecture is flexible and scalable, and describes all of the missions that EMTG can optimize as shown in Figure 10.1

The `Mission` class has the following methods:

- `calcbounds()` - Calculates the upper and lower bounds, as well as scale factors, for any owned decision variables and constraints. Calls the `calcbounds()` methods of any owned `Journey` and `ObjectiveFunction` objects.
- `evaluate()` - Evaluates any owned constraints and their partial derivatives. Calls the `evaluate()` methods of any owned `Journey` and `ObjectiveFunction` objects.
- `output()` - Writes the .emtg output file for the evaluated mission. Calls the `output()` methods of any owned `Journey` and `ObjectiveFunction` objects. Writes the mission-end summary, including all propellant and dry mass information for all spacecraft stages, plus the final decision vector and constraint vector.
- `output_ephemeris()` - Writes an ephemeris file and acceleration output file for the evaluated mission by calling the `output_ephemeris()` of any owned `Journey` objects. Writes a .cmd and .py file that uses `mkspk.exe` to create a SPICE kernel from the ephemeris file.

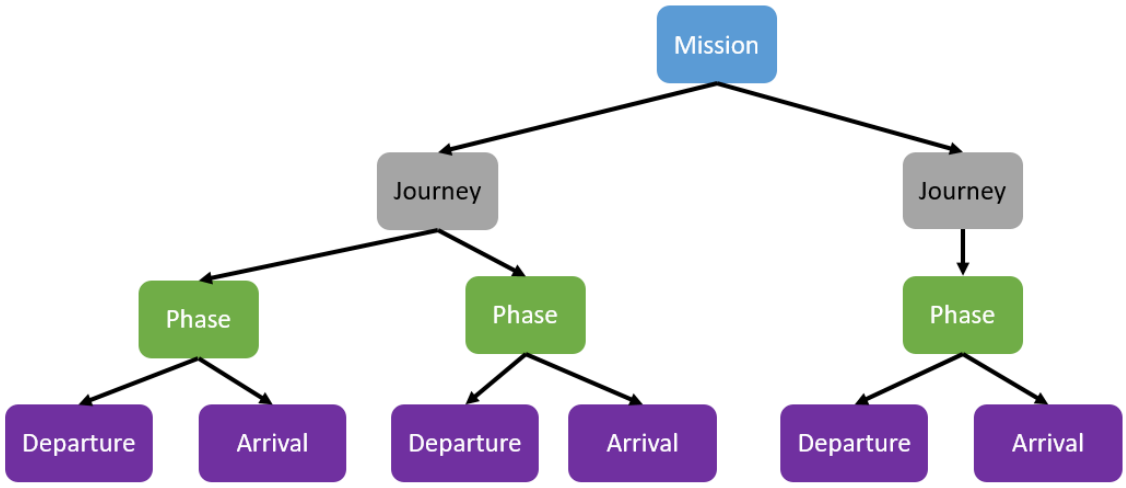


Figure 10.1: Architecture of an EMTG mission.

- `output_STMs()` - Writes text files for each state transition matrix (STM) in the mission by calling the `output_STMs()` method on any owned `Journey` objects.
- `output_maneuver_and_target_spec()` - Writes maneuver and target specification files by calling the `output_maneuver_and_target_spec()` methods of any owned `Journey` objects. These files are read by PIRATE, MONSTER, PyGMATscripter, and eventually also interfaces to Monte and provide the necessary information to re-target each maneuver in an operational navigation tool.
- `construct_initial_guess()` - Constructs the initial guess for the optimization problem. The user-provided `trialX` field in the `missionoptions` structure is compared to the mission `Xdescriptions` vector via string matching. Values from `trialX` are inserted into the initial guess vector when their descriptions match. If state representations do not match, *i.e.* if the user has specified an initial guess in `SphericalRADEC` coordinates but the `missionoptions` file is configured to use `SphericalAZFPA` coordinates, then the user is warned. Low thrust phase initial guesses, whether they be based on two-point shooting or parallel shooting transcriptions, are interpolated to the number of time segments and interior control points requested by the user for that phase. Once all initial guess values are assigned, they are clipped to remain inside the `Xupperbounds` and `Xlowerbounds` vectors that were generated by the constructor. Finally, if any values remain unspecified, `construct_initial_guess` populates the with random values between the allowed bounds.
- `getUnscaledObjective()` - Returns the unscaled value of the objective function.
- `getJourney()` - Returns a pointer to the desired `Journey` object.

In addition, because `Mission` is *not* derived from `sparsey_thing`, `Mission` also implements the following utility methods:

1. `create_sparsity_entry()`
2. `create_sparsity_vector()`

10.2 Mission Time Constraint

The user may impose constraint on total mission flight time. This is simply the sum of all of the time variables in the mission. EMTG locates these by iterating through the `Xdescriptions` vector and adding up all decision variables that contain the string "time."

10.3 Propellant and dry mass constraints

`Mission` may track constraints on chemical fuel, chemical oxidizer, electric, propellant, and/or dry mass. The user may choose to impose these globally, *i.e.* over the entire mission, or at the spacecraft `stage` level.

All propellant usage in EMTG is tracked by means of "virtual tank" variables. The nonlinear computations of true propellant use are book-kept as locally as possible in individual `Phase` and boundary event objects. Each maneuver is assigned a "virtual tank" variable and constraint. The "virtual tank" variable is a free parameter and is constrained to match the computed value of actual propellant use. The total amount of each type of propellant may then be computed by a linear sum of all of the associated virtual tank variables.

The `Spacecraft` object keeps track of the indices in the decision vector corresponding to the virtual tank variables for each propellant type. In addition, each individual `Stage` object keeps track of the indices corresponding to the virtual tank variables of each propellant type assigned to that stage. `Mission`'s only responsibility is to sum up all of the virtual tank variables for each global or stage propellant constraint. The lower bound on each propellant tank constraint is 0.0 kg and the upper bound is the user-defined size of the tank, adjusted by the user-defined propellant margin.

The global and stage dry mass constraints are computed by subtracting the total propellant consumed, plus user-defined margin, from the encoded final mass of the spacecraft. If the spacecraft either dropped mass (*e.g.* dropped a probe) or added mass (*e.g.* picked up a rock from an asteroid), then this mass is also accounted for.

Finally, the user may choose to constrain "final mass" instead of "dry mass." This is just a constraint applied directly to the final state of the spacecraft, ignoring any propellant margin.

Chapter 11

Journey

11.1 Overview

The `Journey` class is an intermediate container that is a member of `Mission` and owns a `boost::ptr_vector` of `Phase` objects. `Journey` inherits from `sparsey_thing` and therefore contains pointers to all of the data vectors that define the NLP problem and all of the utility methods that compute the Jacobian sparsity pattern.

The `Journey` class has the following methods:

- `calcbounds()` - Calculates the upper and lower bounds, as well as scale factors, for any owned decision variables and constraints. Calls the `setup_calcbounds()` and `calcbounds()` methods of any owned `Phase` objects.
- `process_journey()` - Evaluates any owned constraints and their partial derivatives. Calls the `evaluate()` methods of any owned `Phase` objects.
- `output()` - Writes the .emtg output file for the evaluated journey. Calls the `output()` methods of any owned `Phase` objects. Writes the header for each journey, any mass increments, the boundary states, approximated ephemeris-pegged flyby periapse states, and any mass drops. Calls the `output_specialized_constraints()` method on the arrival and departure boundary events of each owned `Phase`.
- `output_ephemeris()` - Writes an ephemeris file and acceleration output file for the evaluated mission by calling the `output_ephemeris()` of any owned `Phase` objects. Writes a .cmd and .py file that uses `mkspk.exe` to create a SPICE kernel from the ephemeris file.
- `output_STMs()` - Writes text files for each state transition matrix (STM) in the mission by calling the `output_STMs()` method on any owned `Phase` objects.
- `output_maneuver_and_target_spec()` - Writes maneuver and target specification files by calling the `output_maneuver_and_target_spec()` methods of any owned `Phase` objects.

These files are read by PIRATE, MONSTER, PyGMATscripter, and eventually also interfaces to Monte and provide the necessary information to re-target each maneuver in an operational navigation tool.

- `getNumberOfPhases()` - Returns the number of owned `Phase` objects.
- `getPhase()` - Returns a pointer to the desired `Phase` object.
- `getFirstPhase()` - Returns a pointer to the first `Phase` object.
- `getLastPhase()` - Returns a pointer to the last `Phase` object.
- `getDeterministicDeltav()` - Sums and returns the deterministic Δv in the owned `Phase` objects. Each `Phase` is responsible for computing its deterministic Δv.
- `getStatisticalDeltav()` - Sums and returns the statistical Δv in the owned `Phase` objects. Each `Phase` is responsible for computing its statistical Δv.
- `getFinalMass()` - Returns the mass at the end of the last `Phase` in this journey.

11.2 Journey-end Δv

The user may define a fixed `journey_end_deltav` for each `Journey`. This could represent, for example, a known set of maneuvers that will be performed after arrival at the target body. This could be expressed as a fixed mass drop, but `journey_end_deltav` allows the user to define a Δv and apply it against the spacecraft mass at the end of the journey, using the appropriate spacecraft `Stage` monoprop system.

The actual computation of propellant use for the journey-end *Deltav* is done in `ArrivalEvent`, but `Journey` extracts that information from the last `Phase` and outputs it.

11.3 Journey time and epoch constraints

The user may elect to constrain a given `Journey`'s departure date. This is done by summing the launch epoch and all of the flight time variables that occur prior to the beginning of the `Journey`.

In addition, the user may choose to constrain a `Journey`'s *flight time*, *aggregate flight time*, or *arrival epoch*. It is not currently possible to constrain more than one of these at a time.

Flight time refers to the sum of all phase time of flight variables and boundary event time widths in the `Journey`. *Aggregate flight time* refers to the sum of all phase flight time and boundary event time width variables up to and including the `Journey`. *Arrival epoch* computed by adding the launch epoch to the *aggregate flight time*.

Chapter 12

Phase

12.1 Phase base class

`Phase` is the abstract base class from which all EMTG phase types derive. The full phase class hierarchy is described in the EMTG Doxygen for the `phase` class. Every phase contains a `DepartureEvent` and an `ArrivalEvent` as described in Chapter 13.

The base `Phase` class is also responsible for appropriately setting the stage of the spacecraft (Section 5.3) based on user-defined values of `stage_after_departure`, `stage_before_arrival`, and `stage_after_arrival`.

`Phase` is responsible for calling the left and right boundaries' `calcbounds` and `process_event` methods. `Phase` also computes the phase's initial TCM, if any, using the spacecraft's monoprop thrusters as described in Section 5.6.1.

When the phase is evaluated, `phase` calls `process_event` on the departure and arrival events to determine the state of the spacecraft after departure and before arrival. Processing of everything that happens between the boundary points is handled by the derived phase classes as described later in this chapter.

`Phase` implements one, the time of flight (TOF) for the phase. The lower and upper bounds of the phase time of flight are determined based on the boundary events as described in `phase::calcbounds_phase_flight_time()`.

`Phase` also implements one optional constraint. If `stage_after_departure` is set, then `phase` constrains the initial mass to be greater than or equal to the current stage dry mass as defined in the spacecraft definition file (Section 5.3).

All phase classes in EMTG define the following methods:

- **setup_calcbounds()** - Sets up pointers to variable holders needed in the calcbounds process.

Performs this task for not only the phase itself but also its owned departure and arrival boundary events.

- **calcbounds()** - Controls the calculation of bounds and sparsity pattern for the phase's decision variables and constraints by calling the below listed calcbounds methods.
- **calcbounds_phase_main()** - Calculates the bounds and sparsity pattern of variables and constraints defined by the derived phase class.
- **calcbounds_virtual_propellant_tanks()** - Calculates the bounds and sparsity pattern of the phase's virtual propellant tank variables and constraints.
- **calcbounds_deltav_contribution()** - Calculates the bounds and sparsity pattern of the phase's contribution to the mission Δv. Only called when the minimum Δv objective function is used.
- **process_phase()** - Evaluates the phase's constraints and contribution to the objective function by calling the below listed process methods.
- **process_phase_main()** - Evaluates the phase transcription. This is always overriden by the derived class.
- **process_phase_left_boundary()** - Calls the phase's departure event's process method and populates any data fields associated with the left-hand side of the phase. Handles the post-departure TCM if applicable. Handled by the base class.
- **process_phase_flight_time()** - Computes the bounds on the phase time of flight. Usually handled by the base class but can be overriden for special phase types. Generally these phase types would still call the base class but then also define additional time-related variables.
- **process_phase_right_boundary()** - Calls the phase's arrival event's process method and populates any data fields associated with the right-hand side of the phase. Handled by the base class.
- **process_virtual_propellant_tanks()** - Computes the virtual propellant constraints, *i.e.* requires that the actual propellant consumed match the virtual propellant.
- **process_deltav_contribution()** - Computes the phase's contribution to the total mission Δv.
- **output()** - Writes lines to the .emtg summary file.
- **output_ephemeris()** - Writes lines to the .ephemeris file. This file may be configured to provide a high-resolution summary of systems parameters (mass, thrust, I_{sp}, *etc.*) or may alternately be written out as just state and epoch and then used to make a SPICE kernel.
- **output_maneuver_and_target_spec()** - Writes lines to the .maneuver_spec and .target_spec files, which are used as an interface to flight-fidelity maneuver planning tools.

12.2 TwoPointShootingPhase

`TwoPointShootingPhase` is an abstract base class for all phase transcriptions that employ two point shooting. In other words, all phases where the spacecraft is propagated forward from the left-hand boundary condition and backward from the right-hand boundary condition. A set of match point constraints is employed to link the forward and backward half-phases together. The derivatives of the match point constraints with respect to the boundary variables may be computed via forward and backward half-phase transition matrix (HPTM)s, which in turn are constructed by chaining together all of the state transition matrix (STM)s and maneuver transition matrix (MTM)s in the forward and backward half-phases.

`TwoPointShootingPhase` handles the creation of the match point constraints and the mapping of the partial derivatives of the boundary states to the match point constraints via the HPTMs. Derivatives of the match point constraints with respect to variables internal to the phase, *i.e.* control variables, are handled by the derived phase class.

12.2.1 CoastPhase

`CoastPhase` is the simplest form of `TwoPointShootingPhase` in which the spacecraft travels from the left boundary to the right boundary without performing any maneuvers. The user may elect to propagate the trajectory via a Kepler propagator or via numerical integration with a higher-fidelity force model. In both cases, the propagation is done in the time domain.

If a numerical integrator is used, the user may independently control the step size on either side of the match point. The user specifies a time step size in seconds for the first (`CoastPhaseForwardIntegrationStepLength`) and second (`CoastPhaseBackwardIntegrationStepLength`) half-phase. The user may also choose `CoastPhaseMatchPointFraction` in $[0, 1]$, the fixed fraction of the phase flight time at which the match point occurs. The default value is 0.5, corresponding to a match point exactly halfway through the phase. A lower number moves the match point closer to the left-hand boundary and a higher number moves the match point closer to the right-hand boundary.

`CoastPhase` adds no new variables or constraints to the problem.

12.2.2 SundmanCoastPhase

`SundmanCoastPhase` is identical to `CoastPhase` except that it uses a Sundman propagator instead of a time-domain propagator. An additional decision variable is added to represent the Sundman anomaly traversed over the course of the phase, and an epoch continuity constraint is added at the match point. `SundmanCoastPhase` is designed for high-fidelity modeling of gravity assists without requiring tight time steps.

`SundmanCoastPhase` is implemented in EMTGv9 with an integrated propagator and provides full analytical derivatives. The independent variable is a scaled Sundman anomaly that roughly

corresponds to eccentric anomaly. EMTGv9 does not yet have a Keplerian Sundman propagator, and so SundmanCoastPhase will not work in Kepler mode. But, since Kepler mode is so fast anyway, you wouldn't want to do that.

12.2.3 MGAnDSMs

EMTG's Multiple Gravity Assist with n Deep-Space Maneuvers (`MGAnDSMs`) transcription models the flight of a spacecraft using high-thrust chemical propulsion. The maneuvers are encoded as impulsive events, *i.e.* they happen instantaneously, and the optimizer may place maneuvers in any permitted location in the phase. The user chooses the maximum number of impulses *a priori*. If the specified number of impulses is more than what is needed, the optimizer will reduce the magnitude of any un-needed maneuvers to zero. `MGAnDSMs` derives from `TwoPointShootingPhase`.

The trajectory is propagated forward in time from the left-hand boundary condition and backward in time from the right-hand boundary condition. The optimizer chooses the time of flight (TOF) for the phase, along with necessary parameters to define the magnitude and direction of any impulsive DSMs. The TOF from the left-hand boundary to the first DSM, as well as from each DSM to the next DSM or to the right-hand boundary where appropriate, is expressed as the product of a "burn index" η_i with the phase TOF. The sum of the η_i must equal 1.0, guaranteeing that the propagation arcs fit within the phase TOF. Therefore, if a phase has only one impulse, then the time from the left boundary to the DSM, Δt_1, and the time from the DSM to the right boundary, Δt_2 will be:

$$\Delta t_1 = \eta_1 TOF \tag{12.1}$$

$$\Delta t_2 = \eta_2 TOF \tag{12.2}$$

Mass is propagated across each impulse by means of the exponential form of the rocket equation as shown in Equation 12.3.

$$m_i^+ = m_i^- \exp\left(\frac{-\Delta v}{I_{sp} g_0}\right) \tag{12.3}$$

where m_i^- is the mass of the spacecraft before the maneuver, m_i^+ is the mass of the spacecraft after the maneuver, Δv is the magnitude of the impulsive DSM, g_0 is the acceleration due to gravity at sea level on Earth, and I_{sp} is the specific impulse of the spacecraft's thruster. A constant mass leak term may be used to approximate propellant consumption due to attitude control system (ACS) desat maneuvers. EMTG keeps track of fuel and oxidizer consumption separately so that they may be individually constrained.

The decision variables and constraints necessary to define an MGAnDSMs phase are listed in Tables 12.1 and 12.2, respectively. A diagram of the MGAnDSMs phase architecture is shown in Figure 12.1. The subscript $_{mp}$ denotes that a given constraint is expressed at the match point.

Each impulse/propagate pair in a `MGAnDSMs` phase is called a `MGAnDSMs_subphase`. `MGAnDSMs_subphase` is also the named of an abstract base class with two derived classes, `Forward_MGAnDSMs_`-

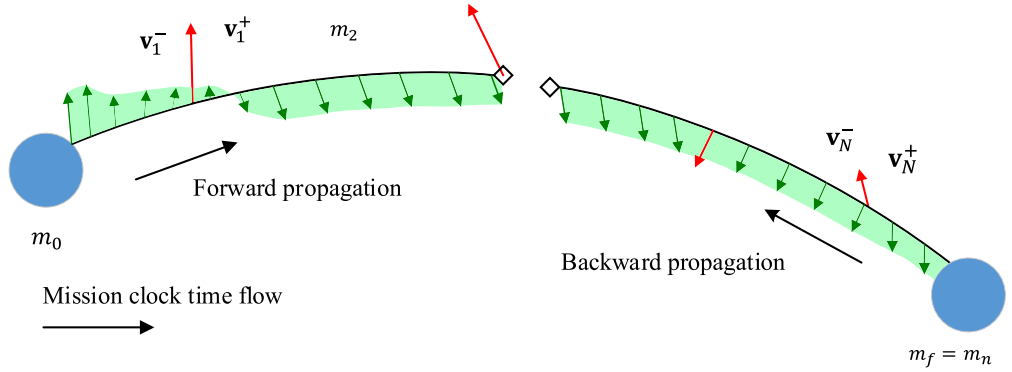


Figure 12.1: Diagram of the MGAnDSMs transcription. The red arrows represent impulsive DSMs, and the green-highlighted black arrows represent natural perturbations.

Table 12.1: Unique variables that define an MGAnDSMs phase

Variable	Description
η_i	Fractions of the phase TOF that define the time between DSMs and the boundaries, as well as between DSMs and other DSMs. One per DSM plus one for the right-hand boundary.
$\Delta v_{i,x}$	x component of DSM i. One per DSM
$\Delta v_{i,y}$	y component of DSM i. One per DSM
$\Delta v_{i,z}$	z component of DSM i. One per DSM
$m_{fuel-virtual}$	virtual chemical fuel tank
$m_{ox-virtual}$	virtual chemical oxidizer tank

Table 12.2: Constraints that define an MGAnDSMs phase.

Constraint	Depends on
$x_{mp}^+ = x_{mp}^-$	TOF, all η_i, all $\Delta v_{i,x}$, all $\Delta v_{i,y}$, all $\Delta v_{i,z}$, boundary variables
$y_{mp}^+ = y_{mp}^-$	TOF, all η_i, all $\Delta v_{i,x}$, all $\Delta v_{i,y}$, all $\Delta v_{i,z}$, boundary variables
$z_{mp}^+ = z_{mp}^-$	TOF, all η_i, all $\Delta v_{i,x}$, all $\Delta v_{i,y}$, all $\Delta v_{i,z}$, boundary variables
$\dot{x}_{mp}^+ = \dot{x}_{mp}^-$	TOF, all η_i, all $\Delta v_{i,x}$, all $\Delta v_{i,y}$, all $\Delta v_{i,z}$, boundary variables
$\dot{y}_{mp}^+ = \dot{y}_{mp}^-$	TOF, all η_i, all $\Delta v_{i,x}$, all $\Delta v_{i,y}$, all $\Delta v_{i,z}$, boundary variables
$\dot{z}_{mp}^+ = \dot{z}_{mp}^-$	TOF, all η_i, all $\Delta v_{i,x}$, all $\Delta v_{i,y}$, all $\Delta v_{i,z}$, boundary variables
$m_{mp}^+ = m_{mp}^-$	TOF, all η_i, all $\Delta v_{i,x}$, all $\Delta v_{i,y}$, all $\Delta v_{i,z}$, boundary variables
$m_{fuel-mp}^+ = m_{fuel-mp}^-$	TOF, all η_i, all $\Delta v_{i,x}$, all $\Delta v_{i,y}$, all $\Delta v_{i,z}$, boundary variables
$m_{ox-mp}^+ = m_{ox-mp}^-$	TOF, all η_i, all $\Delta v_{i,x}$, all $\Delta v_{i,y}$, all $\Delta v_{i,z}$, boundary variables
$\sum_{i=1}^{n+1} \eta_i = 1.0$	all η_i

subphase and Backward_MGAnDSMs_subphase. These two derived classes handle the actual computation of the maneuvers and propagation of the trajectory in the forward and backward sides of the phase, respectively. The parent MGAnDSMs owns vectors of Forward_MGAnDSMs_subphase and Backward_MGAnDSMs_subphase objects, and is responsible for assembling the half-phase transition matrices by chaining the individual subphase state transition matrices.

12.2.3.1 MGAnDSMs maneuver constraints

Each MGAnDSMs_subphase contains a vector of MGAnDSMs_maneuver_constraint objects. MGAnDSMs_maneuver_constraint is an abstract base class from which a wide variety of constraints are derived. Their uses are described in the scripted constraints document. Currently we can constrain maneuver magnitude and epoch both in an absolute sense and relative to other epochs in the phase.

MGAnDSMs_maneuver_constraint objects are constructed by the MGAnDSMs_maneuver_constraint_factory. Whenever a developer adds a new constraint, the factory must be updated.

12.2.4 TwoPointShootingLowThrustPhase

TwoPointShootingLowThrustPhase is an abstract base class that is derived from TwoPointShootingPhase. TwoPointShootingLowThrustPhase is a base class for all transcriptions that model a spacecraft operating with continuous thrust (often low-thrust) propulsion for the entire phase. The user may impose forced initial and/or terminal coasts of fixed duration at the beginning and end of the phase, respectively.

Any remaining flight time not consumed by the forced coasts is available for thrusting. The thrust time is divided up into an even number of control segments, with an equal number of segments on either side of the match point. The segments are assigned equal length in the independent variable of integration. This is usually time but it is possible to construct a derived phase type that integrates over true anomaly or mean anomaly instead of time.

A control 3-vector $\mathbf{u}$ is applied at each control segment. There are $3n$ control variables in a TwoPointShootingLowThrustPhase, where n is the number of control segments. The magnitude of $\mathbf{u}$ represents the commanded duty cycle for the control segment, and is constrained to not exceed 1.0. The user may elect to force the control to have unit magnitude for all control segments, in which case the lower bound is also 1.0, or to force the control to have zero magnitude, in which case the lower and upper bounds are both 0.0. Previous versions of EMTG allowed for a fourth control variable at each step, $u_{command}$. $u_{command}$ was used to control properties of the propulsion system, such as input voltage or mass flow rate. TwoPointShootingLowThrustPhase does not currently support $u_{command}$ but the design allows for it to be added at a later date. The STMs and MTMs would just need an additional row and column. The user may also elect to force the all of the control vectors in a phase to match each other, creating a fixed inertial pointing phase.

TwoPointShootingLowThrustPhase also implements two virtual propellant tank variables. One always represents chemical fuel, whether for main propulsion or for ACS desats. The other rep-

resents electric propellant. These virtual tank variables may be added together to compute a linear constraint on propellant consumption across all phases and boundary events in the mission. Constraints are added to ensure that the virtual propellant variables match the actual propellant consumed in the phase. The latter is highly nonlinear and is handled with extra rows and columns in the STM and MTM in each derived phase class.

As of this writing, EMTG assumes that the spacecraft has single-axis articulated solar arrays. This is important because there is therefore no dependence of available power on thrust direction. Fixed arrays would have cosine losses, which are not yet modeled.

12.2.4.1 MGALT

`MGALTphase` is a low-fidelity transcription for continuous-thrust (often low-thrust) phases. It derives from `TwoPointShootingLowThrustPhase` and models the trajectory using the Sims-Flanagan transcription. The low-thrust perturbation is approximated across each control segment by a bounded impulse, where the magnitude of the impulse is equal to the maximum Δv that could be accomplished by thrusting continuously across the thrust segment. The spacecraft is propagated between impulses by solving Kepler's equation.

The full mathematical details of the MGALT transcription are provided in References [6] and [7]. This section discusses only the software architecture.

The calculations of available power, thrust, and mass flow rate are performed by the spacecraft model and the `BoundedImpulseManeuver` abstract base class. The mathematics of each forward or backward thrust-and-propagation step are performed by the derived `ForwardBoundedImpulseManeuver` and `BackwardBoundedImpulseManeuver` classes, respectively. The computation of derivatives is slightly different between the forward and backward versions and so two classes are necessary.

`ForwardBoundedImpulseManeuver` and `BackwardBoundedImpulseManeuver` track not only the position, velocity, and mass of the spacecraft but also the propellant state. A constant mass leak term may be used to approximate propellant consumption due to ACS desat maneuvers. ACS propellant is applied against the spacecraft's "chemical fuel" tank, whereas main propulsion is applied against the "electric propellant" tank.

`MGALTphase` is responsible for holding data structures, a vector of `ForwardBoundedImpulseManeuver` and `BackwardBoundedImpulseManeuver` calculation objects, and for assembling the forward and backward HPTMs by chaining the STMs and MTMs as described in References [6] and [7]. `MGALTphase` also performs all file output.

The user may choose to constrain the distance between the spacecraft and any reference body at the center of each control segment. This computation is handled by the parent `MGALTphase` class. While effective, this set of constraints significantly slows the optimization process because computing the derivatives of the distance constraints with respect to the phase control variables produces a very dense sparsity pattern and requires many matrix multiplies.

12.2.4.2 FBLT

Finite-Burn Low-Thrust (FBLT) is identical to MGALT in every way except that instead of the Sims-Flanagan transcription, FBLT uses explicit numerical integration to propagate the equations of motion with a thrust perturbation supplied by the electric propulsion and power system models (Chapter 5). Natural perturbations may also be added as described in Section 4.7.

`FBLTphase` derives from `TwoPointShootingLowThrustPhase` and adds no new decision variables or constraints. The user may elect to constrain the distance between the spacecraft and a reference body, as described in the scripted constraints document. As in MGALT, the sparsity pattern of the distance constraint is a triangle and requires many STM multiplications, and so the distance constraint adds significantly to the computation time necessary to optimize the mission.

12.2.5 MGALTS

Multiple Gravity Assist with Low-Thrust using the Sundman transformation (MGALTS) is not yet implemented, but Donald and Jacob built a demonstrator once and will some day put it into EMTGv9 if anyone can figure out a use for it. It is just MGALT with a Sundman propagator, an additional decision variable for the Sundman anomaly traversed by the phase, and an epoch continuity constraint.

MGALTS uses a Keplerian Sundman propagator that is not yet implemented. Again, we'll build this if and when someone needs it. We did this in EMTGv8 and it worked but was not terribly useful. This subsection is a placeholder for now.

12.2.6 FBLTS

Finite-Burn Low-Thrust using the Sundman transformation (FBLTS) is not yet implemented, but Donald and Jacob built a demonstrator once and will some day put it into EMTGv9 if anyone can figure out a use for it. It is just FBLT with a Sundman propagator, an additional decision variable for the Sundman anomaly traversed by the phase, and an epoch continuity constraint.

FBLTS uses an integrated Sundman propagator. Again, we'll build this if and when someone needs it. We did this in EMTGv8 and it worked but was not terribly useful. This subsection is a placeholder for now.

12.3 Parallel shooting phase classes

`ParallelShootingPhase` is an abstract base class for phase transcriptions that model the path of a spacecraft using low-thrust propulsion via the method of direct parallel shooting [8]. In this technique, the phase is decomposed into a set of short shooting steps. The state vector is encoded

as decision variables on the left-hand side of each short step and then propagated to the right-hand side. A set of nonlinear match point constraints are used to ensure that in a converged solution, the propagated right-hand side of the $(i-1)^{th}$ step matches the encoded left-hand side of the i^{th} step. The constraints are encoded on the left-hand side of the i^{th} step. The technique is called "parallel shooting" because all of the steps propagate in the same direction, *i.e.* forward in time.

The primary role of `ParallelShootingPhase` is to hold a polymorphic vector of `ParallelShootingStep`-derived objects as described in the next paragraph. In addition, `ParallelSootingPhase` handles the forced initial and/or terminal coast if the user requests them. The terminal coast is backward propagated, unlike everything else in `ParallelShootingPhase`.

`ParallelShootingStep` is the abstract base class for a propagation step in `ParallelShootingPhase`. It handles the encoding of the left-hand side state vector and the match point constraints that connect to the right-hand side of the previous step. As of this writing, `ParallelShootingStep` can encode the state vector in `SphericalRADEC` or `SphericalAZFPA` coordinates. The match point constraints are always encoded in cartesian coordinates. Additional state and constraint representations can easily be added. The phase type-specific derived classes of `ParallelShootingStep` are also responsible for mapping the partial derivatives of the left-hand state to the right-hand state by means of STMs and MTMs that are unique to that phase type. In addition to position and velocity, `ParallelShootingStep` also encodes decision variables and continuity constraints for spacecraft mass, virtual chemical fuel mass, and virtual electric propellant mass.

There are three abstract base classes that themselves derive from `ParallelShootingStep`. `ParallelShootingFirstStep` is the abstract base class for the first step in a `ParallelShootingPhase`. It is unique in that instead of retrieving right-hand propagated state information from the previous step, it pulls the state after the initial coast from the parent `ParallelShootingPhase`. `ParallelShootingLastStep` is the abstract base class for the last step in a `ParallelShootingPhase` and is unique because it has two sets of match point constraints - the usual one on the left-hand side and another set on the right-hand side to ensure that the propagated right-hand state matches the `ParallelShootingPhase`'s state prior to the terminal coast. Finally, `ParallelShootingOneStepToRuleThemAll` is an abstract base class for the special case where a `ParallelShootingPhase` has only one step and has the properties of both `ParallelShootingFirstStep` and `ParallelShootingLastStep`.

12.3.1 PSFB

PSFB is a transcription for modeling the path of a low-thrust spacecraft using direct parallel shooting and a high-fidelity model of both the natural and spacecraft dynamics. PSFB consists of the `PSFBphase`, `PSFBstep`, `PSFBfirststep`, `PSFBlaststep`, and `PSFBOneStepToRuleThemAll` classes that are each derived from the corresponding base class in Section 12.3, as well as a `PSFBstep_factory` that creates the various types of step.

PSFB propagates the trajectory using explicit numerical integration as described in Chapter 7 and EMTG's full dynamics and spacecraft model as described in Chapters 4 and 5. Just as in EMTG's other transcriptions, control is piecewise constant across a segment. The user selects to have one *or more* control opportunities per segment. If more than one control opportunity is chosen

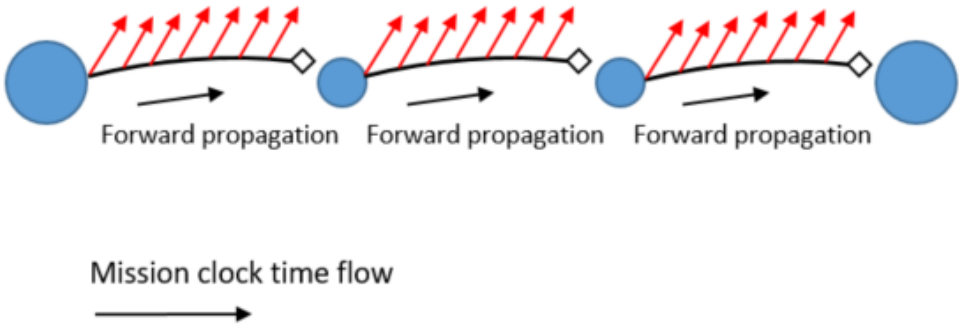


Figure 12.2: Diagram of the PSFB transcription. The red arrows represent a continuous thrust perturbation.

then the segment is broken into equal-length control steps. Control is encoded as a 3-vector just as in MGALT and FBLT. There is currently no provision for a fourth control variable but it would not be too difficult to add one. The user may elect to force the all of the control vectors in a phase to match each other, creating a fixed inertial pointing phase. The user may also elect to force the control to be either full on or full off across the phase. Figure 12.2 shows a PSFB phase.

The PSFB derived classes are responsible only for propagation, calculation of STMs, and output. All of the book-keeping is handled by the base classes.

12.3.1.1 PSFB with high-fidelity duty cycle

PSFB, like FBLT, models propulsion duty cycle by multiplying the maximum duty cycle by the commanded thrust and mass flow rate. While this averaged duty cycle is sufficient for trade studies and sensitivity analysis, it may not always be acceptable as an initial guess for a flight-fidelity maneuver planning tool. EMTG provides an alternative form of the PSFB transcription that explicitly models the individual thrust arc and the short coast that exist in each thrust segment. The coast period is set aside for tracking, communication, other mission needs (*e.g.* optical navigation), and contingencies. As with all of EMTG's other low-thrust transcriptions, power and propulsion characteristics are continuously evaluated as the trajectory is propagated and so thruster mode transitions can occur during a segment. Figure 12.3 shows three PSFB steps with high-fidelity duty cycle modeling and thrust transitions due to a high-fidelity propulsion model.

The high-fidelity duty cycle variant of PSFB is implemented by creating yet another set of de-

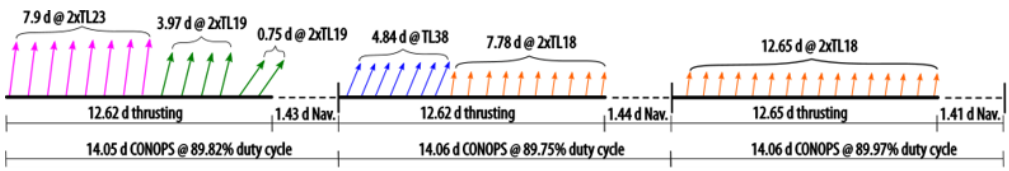


Figure 12.3: Diagram of a three PSFB steps with high-fidelity duty cycle modeling.

rived classes: `PSFB_HifiDuty_step`, `PSFB_HifiDuty_firststep`, `PSFB_HifiDuty_laststep`, and `PSFB_HifiDuty_OneStepToRuleTHemAll`. No new phase class is necessary because the container of `PSFBstep` objects in `PSFBphase` is polymorphic and can be populated with the high-fidelity duty cycle variants instead.

12.3.2 PSBI

Parallel Shooting with Bounded Impulses (PSBI) is a low-fidelity parallel-shooting transcription that combines the Sims-Flanagan model [9] with the base parallel-shooting phase classes. In PSBI, the low-thrust acceleration is modeled as a bounded impulse in the center of each time step just as in MGALT. Just as in PSFB, each time-step encodes its left-hand state in the decision vector and includes a set of continuity constraints to ensure that the encoded left-hand state matches the previous step's propagated right-hand state. Figure 12.4 describes the PSBI transcription.

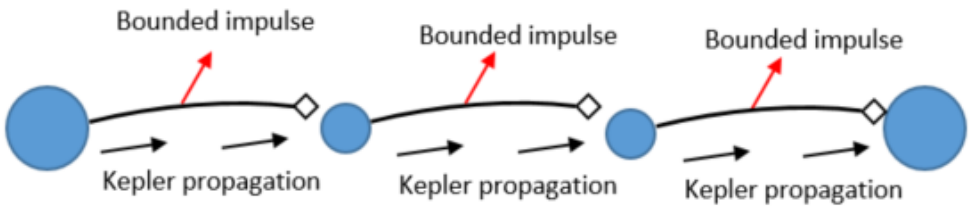


Figure 12.4: The PSBI transcription.

PSBI is composed of the `PSBIphase` container class, the `PSBIstep` base class for time steps, and the `PSBIfirststep`, `PSBIlaststep`, and `PSBIOneStepToRuleThemAll`. Each of these is derived from the corresponding abstract base class for `ParallelShootingPhase` as described in Section 12.3. The PSBI step classes are constructed by `PSBIstep_factory`.

PSBI uses exactly the same decision variables and constraints as PSFB, as defined in the base `ParallelShootingPhase` and child classes. Since PSBI runs much faster than PSFB, this enables the user to quickly converged a solution in PSBI and then use it as an initial guess for PSFB.

12.3.3 Parallel Shooting Constraints

The base `ParallelShootingStep` class contains vectors of `ParallelShootingStepDistanceConstraint` and `ParallelShootingStep_maneuver_constraint` objects. These constraints are enforced on the left-hand side of each segment. `ParallelShootingStepDistanceConstraint` is a stand-alone class that there is no need to inherit from as far as we can think of. Its use is described in the scripted constraints document.

`ParallelShootingStep_maneuver_constraint` is an abstract base class that is designed so that a developer may easily write derived classes to constraint maneuver magnitude, direction, etc. As of this writing (12/27/2019), there is only one such derived constraint, the Body-Probe-Thrust (BPT) angle constraint. This constraint is described in detail in the scripted constraints document.

Maneuver constraints are the primary reason to use PSFB and PSBI instead of FBLT and MGALT. Maneuver constraints are very difficult to pose in a two-point shooting transcription but very easy in a parallel shooting transcription because the state vector at the left-hand side of each segment is encoded into the decision vector and no chaining is necessary.

`ParallelShootingStep_maneuver_constraint` objects are constructed by the `ParallelShootingStep_maneuver_constraint_factory`.

Chapter 13

Boundary Conditions

13.1 BoundaryEventBase

`BoundaryEventBase` is the abstract base class from which all boundary events are derived. The base class defines the interfaces common to all boundary events and also the common fields, including the state before and after the event. For a full class hierarchy of all boundary conditions, see "EMTG::BoundaryEvents::BoundaryEventBase" in the EMTG Doxygen. All classes are described conceptually below.

13.2 EdelbaumSpiral

`EdelbaumSpiral` is an owned subclass of `EphemerisPeggedSpiralDeparture` and `EphemerisPegged-SpiralArrival` and handles the mathematics of a segmented Edelbaum Spiral. This technique is used in EMTG to approximate a many-revolution low-thrust spiral about a body in the current universe. For example, one may wish to spiral from LEO to escape from the Earth, or from the edge of the Mars sphere of influence down to the orbital distance of Phobos or Deimos. Edelbaum's approximation provides a fast, sufficiently accurate model that allows spirals to be included with an EMTG broad search without having to explicitly model and optimize the path of the spacecraft during the spiral.

Edelbaum's approximation consists of modeling the initial and final orbits about the body as co-planar circles and then assuming that the thrust level is sufficiently low that the transfer orbit is also nearly circular. The total Δv for the transfer may then be written as:

$$\Delta v = |\sqrt{\frac{\mu}{r_1}} - \sqrt{\frac{\mu}{r_2}}| \tag{13.1}$$

Edelbaum's technique makes the assumption of constant thrust and specific impulse across

Table 13.1: Decision variables that define an `EdelbaumSpiralSegment`

Variable	Description
TOF	time-of-flight of the segment
m_f	mass at the right-hand side of the segment
m_{ep}	virtual electric propellant
m_{cf}	virtual chemical fuel (for ACS desats)

Table 13.2: Constraints that define an `EdelbaumSpiralSegment`

Constraint	Depends on
$TOF_{computed} = TOF_{encoded}$	$TOF_{encoded}$, all previous time variables, m_i
$m_{ep,computed} = m_{ep,encoded}$	$m_{ep,encoded}$, all previous time variables, m_i
$m_{cf,computed} = m_{cf,encoded}$	$m_{cf,encoded}$, m_i (if tracking ACS), $TOF_{encoded}$ (if tracking ACS)

the entire spiral. This does not accurately reflect a spiral using solar-electric propulsion about a body whose orbit has significant eccentricity because as the body moves closer to or farther from the sun, the available power changes and so does the available thrust and specific impulse. Accordingly, EMTG supports splitting an Edelbaum spiral into multiple segments, each modeling an equal portion of the spiral Δv. At the beginning of each segment, the power, thrust, and specific impulse are re-computed based on the body's distance from the sun.

The actual mathematics of the Edelbaum segments are handled in `EdelbaumSpiralSegment`. `EdelbaumSpiral` contains a vector of `EdelbaumSpiral` objects. Other than holding the segment objects, `EdelbaumSpiral`'s job is to calculate the total Δv as per Equation 13.1 and to provide the final state, *i.e.* the state at the end of the final `EdelbaumSpiralSegment`, and its derivatives to the parent boundary condition object. `EdelbaumSpiral` does not itself introduce any new decision variables or constraints.

13.2.1 EdelbaumSpiralSegment

`EdelbaumSpiralSegment` is an owned subclass of *EdelbaumSpiral* that performs the actual Edelbaum spiral calculations for each segment. The segment Δv is provided at problem setup via Equation 13.1 and is fixed during problem execution. In order to simplify the partial derivatives of the spiral's final state with respect to the decision variables, EMTG uses a "sparse" spiral transcription.

The time of flight, the mass at the end of the segment, and the electric propellant and chemical fuel (for ACS) consumed during the segment are encoded as decision variables as described in Table 13.1. Nonlinear constraints are then imposed as per Table 13.2 to ensure that the encoded flight time, mass, and propellant match the computed flight time, mass, and propellant. These constraints are simple to formulate and have simple partial derivatives. EMTG can then compute the total propellant and time required for the spiral by simply adding up the relevant decision variables for each `EdelbaumSpiralSegment`.

The initial mass for each segment, m_i, is drawn either from the parent boundary condition's

encoded mass (for the first segment) or from the previous segment's encoded final mass (for later segments).

13.3 DepartureEvent

`DepartureEvent` is an intermediate abstract base class that specializes `BoundaryEventBase` to the specific needs of a departure event. `DepartureEvent` is then an abstract base class for all of the various departure events described below.

The child departure events all derive both from `DepartureEvent` and from whatever class of boundary they are (`EphemerisPeggedboundary`, `EphemerisReferencedBoundary`, `FreePoint-Boundary`, or `PeriapseBoundary`).

Some departure events encode a "wait time," *i.e.* a decision variable that defines the period of time between either the `launch_window_open_date` or the end of the previous journey, and the beginning of the current journey. The bounds on the wait time are defined by the user. Any departure event that has a wait time and begins a journey other than the first includes a mass continuity constraint such that the mass at the beginning of the departure event matches the mass at the end of the previous journey's arrival event.

13.4 ArrivalEvent

`ArrivalEvent` is an intermediate abstract base class that specializes `BoundaryEventBase` to the specific needs of a arrival event. `ArrivalEvent` is then an abstract base class for all of the various arrival events described below.

The child arrival events all derive both from `ArrivalEvent` and from whatever class of boundary they are (`EphemerisPeggedboundary`, `EphemerisReferencedBoundary`, `FreePointBoundary`, or `PeriapseBoundary`).

13.5 EphemerisPeggedBoundary

`EphemerisPeggedBoundary` is an abstract base class for all boundary events that are pegged to an ephemeris object. For example, if you want to intercept or rendezvous with Ceres, or depart from Ceres in a "patched-conic" fashion, all of those boundary events are "ephemeris-pegged." The base `EphemerisPeggedBoundary` class contains the code to look up the position and velocity of the ephemeris point as a function of time and insert it into the spacecraft state vector.

The user provides the identity of the ephemeris object to which the boundary is pegged. The position and velocity of the body, along with derivatives, will then be computed with either SPICE or SplineEphem (Chapter 8).

13.5.1 EphemerisPeggedDeparture

`EphemerisPeggedDeparture` derives from both `EphemerisPeggedBoundary` and `DepartureEvent`, and is an abstract base class for several boundary classes as defined below.

13.5.1.1 EphemerisPeggedFreeDirectDeparture

`EphemerisPeggedFreeDirectDeparture` is the simplest form of `EphemerisPeggedDeparture` in which the spacecraft takes on the position and velocity of the ephemeris point. Mass is chosen as a decision variable. `EphemerisPeggedFreeDirectDeparture` has a wait time, and therefore in journeys after than the first, its `DepartureEvent` base class will create a mass continuity constraint. If the `EphemerisPeggedFreeDirectDeparture` is the first event in the mission, then the user may choose to either fix the mass (by setting `allow_initial_mass_to_vary` to false), or allow the mass to vary between zero and the user-defined `maximum_mass` (by setting `allow_initial_mass_to_vary` to true).

13.5.1.2 EphemerisPeggedLaunchDirectInsertion

`EphemerisPeggedLaunchDirectInsertion` describes a patched-conic launch or departure event. Three new decision variables are added - the magnitude of the departure v_∞, and the right ascension and declination of the departure asymptote in the ICRF. The user provides the bounds for all three decision variables.

If the `EphemerisPeggedLaunchDirectInsertion` begins the first journey in the mission, then the mass is chosen according to the launch vehicle model (Section 5.2). The user may select from a range of launch vehicle models including "fixed initial mass." The user may also choose to either fix the initial mass to whatever the launch vehicle model provides for a given value of v_∞ (by setting `allow_initial_mass_to_vary` to false), or allow the mass to vary between zero and the maximum value provided by the launch vehicle (by setting `allow_initial_mass_to_vary` to true). Each launch vehicle model comes complete with minimum and maximum C_3 values, and if they

are more restrictive than the user-defined bounds on v_∞, then EMTG will adjust those bounds to fit within the launch vehicle's capability. If the launch vehicle's capability to a given v_∞ exceeds the user-defined `maximum_mass`, then the mass will be truncated to the user-defined value.

If the `EphemerisPeggedLaunchDirectInsertion` begins a later journey in the mission, then the departure maneuver will be modeled as an impulsive burn using the spacecraft's thrusters.

13.5.1.3 EphemerisPeggedFlybyOut

`EphemerisPeggedFlybyOut` is an abstract base class for all of the types of `EphemerisPeggedDeparture` that represent the outgoing half of a patched-conic flyby - `EphemerisPeggedZeroTurnFlyby`, `EphemerisPeggedUnpoweredFlyby`, and `EphemerisPeggedPoweredFlyby`. `EphemerisPeggedFlybyOut` encodes three new decision variables, the x, y, and z components of $\mathbf{v}_{\infty-out}$ in the ICRF. `EphemerisPeggedFlybyOut`-derived boundary events pull their mass directly from the end of the previous journey and so do not encode their own mass variable. They do, however, all need to do math based on the previous arrival event's $\mathbf{v}_{\infty-in}$ and therefore `EphemerisPeggedFlybyOut` locates decision variables that define that vector.

13.5.1.4 EphemerisPeggedZeroTurnFlyby

`EphemerisPeggedZeroTurnFlyby` is the simplest form of ephemeris-pegged outgoing flyby. This boundary event is used when the flyby is of a very small body and therefore the bend angle is very small. Because the derivatives of a very small bend angle are highly unstable, it is both adequate and recommended to simply not model the bend angle at all. `EphemerisPeggedZeroTurnFlyby` therefore includes three equality constraints to guarantee that $\mathbf{v}_{\infty-out}$ matches the previous event's $\mathbf{v}_{\infty-in}$.

13.5.1.5 EphemerisPeggedUnpoweredFlyby

`EphemerisPeggedUnpoweredFlyby` defines the outgoing half of a patched conic flyby about a body large enough to generate a significant bend angle. The flyby is unpowered, *i.e* no maneuver is performed at periapse.

`EphemerisPeggedUnpoweredFlyby` adds two new constraints - one to require that the magnitude of $v_{\infty-out}$ matches the magnitude of the previous event's $v_{\infty-in}$, and one to ensure that the bend angle does not require the spacecraft to fly below a user-defined safe distance h_{safe} from the body as described in Equations 13.2-13.4.

$$F = h_{FB} - h_{safe} \tag{13.2}$$

$$h_{FB} = \frac{\mu}{v_{\infty-out}^2}\left(\frac{1}{\sin\frac{\delta_{FB}}{2}} - 1\right) - r_{body} \tag{13.3}$$

$$\delta_{FB} = \arccos\left(\frac{\mathbf{v}_{\infty-out} \bullet \mathbf{v}_{\infty-in}}{v_{\infty-out} v_{\infty-in}}\right) \tag{13.4}$$

13.5.1.6 EphemerisPeggedPoweredFlyby

`EphemerisPeggedPoweredFlyby` defines the outgoing half of a patched conic flyby about a body large enough to generate a significant bend angle. The flyby is powered, *i.e* an impulse is performed at periapse aligned with the spacecraft's velocity vector.

`EphemerisPeggedPoweredFlyby` adds one new variable, defining the periapse distance r_p, and one constraint to require that the bend angle be feasible as described below.

$$F = \arcsin\frac{1}{e_{in}} + \arcsin\frac{1}{e_{out}} - \delta_{FB} \tag{13.5}$$

$$\delta_{FB} = \arccos\left(\frac{\mathbf{v}_{\infty-out} \bullet \mathbf{v}_{\infty-in}}{v_{\infty-out} v_{\infty-in}}\right) \tag{13.6}$$

$$e_{in} = 1 + \mathbf{v}_{\infty-in} \bullet \mathbf{v}_{\infty-in}\frac{r_p}{\mu} \tag{13.7}$$

$$e_{out} = 1 + \mathbf{v}_{\infty-out} \bullet \mathbf{v}_{\infty-out}\frac{r_p}{\mu} \tag{13.8}$$

In addition, `EphemerisPeggedPoweredFlyby` must compute the Δv magnitude of the periapse impulse and also the change in mass due to the maneuver. The *Deltav* calculation is described as,

$$\Delta v_{FB} = |B_{\Delta v} - A_{\Delta v}| \tag{13.9}$$

$$A_{\Delta v} = \sqrt{\mathbf{v}_{\infty-in} \bullet \mathbf{v}_{\infty-in} + 2\frac{mu}{r_p}} \tag{13.10}$$

$$B_{\Delta v} = \sqrt{\mathbf{v}_{\infty-out} \bullet \mathbf{v}_{\infty-out} + 2\frac{mu}{r_p}} \tag{13.11}$$

and the mass calculation is described as,

$$m_{after-flyby} = m_{before_flyby} - m_{fuel} - m_{oxidizer} \tag{13.12}$$

where m_{fuel} and $m_{oxidizer}$ are computed using the chemical propulsion model as described in Section **??**.

13.5.1.7 EphemerisPeggedSpiralDeparture

`EphemerisPeggedSpiralDeparture` models an escape spiral from a body in the universe. The user provides the starting and ending orbit radius. `EphemerisPeggedSpiralDeparture` contains an `EdelbaumSpiral` object that does all of the computations. `EphemerisPeggedSpiralDeparture` puts the final state and its derivatives into the standard `state_after_event`, `Derivatives_of_StateAfterEvent`, and `Derivatives_of_StateAfterEvent_wrt_Time`.

13.5.2 EphemerisPeggedArrival

`EphemerisPeggedArrival` derives from both `EphemerisPeggedBoundary` and `ArrivalEvent`, and is an abstract base class for several boundary classes as defined below.

13.5.2.1 EphemerisPeggedLTRendezvous

`EphemerisPeggedLTRendezvous` is the simplest form of `EphemerisPeggedArrival` in which the spacecraft takes on the position and velocity of the ephemeris point. The spacecraft mass is chosen between ±1.0e-13 and the user-defined `maximum_mass`.

13.5.2.2 EphemerisPeggedArrivalWithVinfinity

`EphemerisPeggedArrivalWithVinfinity` is an abstract base class for all `EphemerisPeggedArrival` events that require a $\mathbf{v}_\infty$ vector. `EphemerisPeggedArrival` encodes three new decision variables, the x, y, and z components of $\mathbf{v}_{\infty-in}$ in the ICRF.

13.5.2.3 EphemerisPeggedChemRendezvous

`EphemerisPeggedChemRendezvous` is a derived class of `EphemerisPeggedArrivalWithVinfinity` that represents the case where the spacecraft performs an impulsive maneuver to match both position and velocity with the target body but ignores the gravity of that body. This is suitable for modeling rendezvous with an asteroid or comet. The rendezvous maneuver performance is calculated using the chemical propulsion model in Section **??**.

13.5.2.4 EphemerisPeggedOrbitInsertion

`EphemerisPeggedOrbitInsertion` is a derived class of `EphemerisPeggedArrivalWithVinfinity` that represents the case where the spacecraft performs a two-dimensional patched-conic orbit insertion at the target body. By "two-dimensional," we mean that EMTG only considered the

semi-major axis (SMA) and eccentricity (ECC) of the desired orbit about the target, and none of the angles. The magnitude of the insertion Δv is computed as,

$$\Delta v = v_{p-hyperbola} - v_{p-ellipse} \tag{13.13}$$

$$v_{p-ellipse} = \sqrt{\mu * \left(\frac{2}{r_p} - \frac{1}{SMA}\right)} \tag{13.14}$$

$$v_{p-hyperbola} = \sqrt{v_\infty^2 + 2\frac{mu}{r_p}} \tag{13.15}$$

$$r_p = SMA\,(1 - ECC) \tag{13.16}$$

The user may elect to perform a fixed-magnitude TCM immediately prior to the insertion maneuver, which is calculated using the chemical propulsion model in Section **??**.

13.5.2.5 EphemerisPeggedFlybyIn

`EphemerisPeggedFlybyIn` is a derived class of `EphemerisPeggedArrivalWithVinfinity`. `EphemerisPeggedFlybyIn` adds the ability for the user to define bounds for the components of $\mathbf{v}_{\infty-in}$ and to model a TCM, whose performance is calculated using the chemical propulsion model in Section **??**.

`EphemerisPeggedFlybyIn` is also a base class of `EphemerisPeggedIntercept`. If a patched conic phase appears in the middle of a journey, then `EphemerisPeggedFlybyIn` is used. If the event occurs at the end of a journey, then *EphemerisPeggedIntercept* is used instead.

13.5.2.6 EphemerisPeggedIntercept

`EphemerisPeggedIntercept` represents the scenario where the spacecraft intercepts a body at the end of a journey, *i.e.* matches position but not velocity. `EphemerisPeggedIntercept` is derived from `EphemerisPeggedFlybyIn` but adds the ability to constrain the magnitude of $v_{\infty-in}$.

13.5.2.7 EphemerisPeggedMomentumTransfer

`EphemerisPeggedMomentumTransfer` is a derived class of `EphemerisPeggedIntercept` that is used in the special case where the spacecraft collides with the destination body and transfers momentum to it. The state after the event therefore represents the destination body after the collision and is represented by:

$$m^+ = m_{s/c} + m_{body} v_z^+ \qquad = v_{z-body} + v_{\infty-z-s/c}\beta\frac{m_{s/c}}{\left(m_{s/c}+m_{body}\right)} \tag{13.17}$$

$$v_y^+ = v_{y-body} + v_{\infty-y-s/c}\beta\frac{m_{s/c}}{\left(m_{s/c}+m_{body}\right)} \tag{13.18}$$

$$v_x^+ = v_{x-body} + v_{\infty-x-s/c}\beta\frac{m_{s/c}}{\left(m_{s/c}+m_{body}\right)} \tag{13.19}$$

$$z^+ = z_{body} \tag{13.20}$$

$$y^+ = y_{body} \tag{13.21}$$

$$x^+ = x_{body} \tag{13.22}$$

$$\tag{13.23}$$

The term β is a scale factor that encompasses the plasticity of the impact, the crater formation, and the ejecta released by the impact. The user specifies β in the `JourneyOptions` object as `impact_momentum_enhancement_factor`.

13.5.2.8 EphemerisPeggedSpiralArrival

`EphemerisPeggedSpiralArrival` models a capture spiral from a body in the universe. The user provides the starting and ending orbit radius. `EphemerisPeggedSpiralArrival` contains an `EdelbaumSpiral` object that does all of the computations. `EphemerisPeggedSpiralArrival` puts the final state and its derivatives into the standard `state_after_event`, `Derivatives_of_StateAfterEvent`, and `Derivatives_of_StateAfterEvent_wrt_Time`.

13.6 EphemerisReferencedBoundary

`EphemerisReferencedBoundary` is the abstract base class for all boundary events that are defined *relative* to an ephemeris point but not *on* the ephemeris point. In other words, the boundary point is "referenced" to an ephemeris point and moves with it, but additional information is needed to define the boundary relative to the ephemeris point.

In EMTGv9, ephemeris-referenced boundary conditions are defined as lying on a triaxial ellipsoid centered on an ephemeris point. For example, this could include the sphere of influence of a planet or a triaxial ellipsoid representing the surface of a non-spherical body like Ceres. The user provides the three semi-axes of the ellipsoid in the ICRF coordinate system. The `EphemerisReferencedBoundary` base class then creates two new variables to represent the ICRF right-ascension and declination of the boundary point's position on the ellipsoid. The distance from the center of the ellipsoid is then computed as,

$$r = \sqrt{\frac{1.0}{\frac{\cos^2 RA \cos^2 DEC}{a^2} + \frac{\sin^2 RA \cos^2 DEC}{b^2} + \frac{\sin^2 DEC}{c^2}}} \tag{13.24}$$

`EphemerisReferencedBoundary` also computes the partial derivatives of the boundary point relative to the RA, and DEC decision variables as well as any variables that affect the position of the ephemeris point that the boundary is referenced to. The velocity of the boundary point relative to the ephemeris point is assumed to be zero unless overriden by a derived class, *e.g.* as described in Sections 13.6.9 and 13.6.13 below.

13.6.1 EphemerisReferencedDeparture

`EphemerisReferencedDeparture` derives from both `EphemerisReferencedBoundary` and `DepartureEvent`, and is an abstract base class for several boundary classes as defined below. Typically these classes are only used to define the first boundary event in a mission.

13.6.2 EphemerisReferencedDepartureExterior

`EphemerisReferencedDepartureExterior` derives from `EphemerisReferencedDeparture`, and represents the case where the boundary point lies on the edge of a triaxial ellipsoid surrounding a *body in the current journey's universe*. The spacecraft can then be thought of as *exiting* the ellipsoid. Relevant examples include a low-thrust spiral escape, where the spiral itself is not modeled in EMTG. `EphemerisReferencedDepartureExterior` is an abstract base class for several boundary classes as defined below.

13.6.3 EphemerisReferencedFreeDirectDepartureExterior

`EphemerisReferencedFreeDirectDepartureExterior` is a derived class of `EphemerisReferencedDepartureExterior` that represents a spacecraft beginning at the boundary of a Universe sphere of influence (SOI) and traveling inward. It encodes no new variables or constraints.

13.6.4 EphemerisReferencedDepartureInterior

`EphemerisReferencedDepartureExterior` derives from `EphemerisReferencedDeparture`, and represents the case where the boundary point lies on the edge of a triaxial ellipsoid surrounding the *central body of the current journey's universe*. The spacecraft can then be thought of as *entering* the ellipsoid. Relevant examples include a low-thrust capture at a body, where the interplanetary trajectory is not modeled in EMTG. `EphemerisReferencedDepartureExterior` is an abstract base class for several boundary classes as defined below.

13.6.5 EphemerisReferencedFreeDirectDepartureInterior

`EphemerisReferencedFreeDirectDepartureInterior` is a derived class of `EphemerisReferencedDepartureInterior` that represents a spacecraft beginning at the boundary of a body's SOI and traveling outward. It encodes no new variables or constraints.

13.6.6 EphemerisReferencedArrival

`EphemerisReferencedArrival` derives from both `EphemerisReferencedBoundary` and `ArrivalEvent`, and is an abstract base class for several boundary classes as defined below.

13.6.7 EphemerisReferencedArrivalExterior

`EphemerisReferencedArrivalExterior` derives from `EphemerisReferencedArrival`, and represents the case where the boundary point lies on the edge of a triaxial ellipsoid surrounding a *body in the current journey's universe*. The spacecraft can then be thought of as *entering* the ellipsoid from the *exterior*. Relevant examples include entering the sphere of influence of a body or landing on the surface of a body. `EphemerisReferencedArrivalExterior` is an abstract base class for several boundary classes as defined below.

On the right-hand side of the boundary, the state vector is transformed by *subtracting* the position and velocity of the ephemeris point relative to the central body, thus transforming the state into the frame of the ephemeris point.

13.6.8 EphemerisReferencedLTRendezvousExterior

`EphemerisReferencedLTRendezvousExterior` is the simplest form of `EphemerisReferencedArrivalExterior` in which the spacecraft comes to a rest relative to the ephemeris point at the edge of the bounding ellipsoid. `EphemerisReferencedLTRendezvousExterior` does not include any additional decision variables or constraints.

13.6.9 EphemerisReferencedArrivalWithVinfinityExterior

`EphemerisReferencedArrivalWithVinfinityExterior` extends `EphemerisReferencedArrivalExterior` by adding three decision variables for the magnitude, right ascension, and declination of the velocity vector in the ICRF. `EphemerisReferencedArrivalWithVinfinityExterior` serves as an abstract base class for ephemeris-referenced "exterior" boundary events that require a velocity vector relative to the bounding ellipsoid. The user defines the bounds on the velocity magnitude.

13.6.10 EphemerisReferencedInterceptExterior

`EphemerisReferencedInterceptExterior` is a derived class of `EphemerisReferencedArrivalWithVinfinityExterior` that describes a spacecraft arriving at the bounding ellipsoid with a relative velocity and an optional impulsive TCM. The user defines the size of the TCM and the performance is calculated from the spacecraft's monoprop system as defined in Chapter 5.

13.6.11 EphemerisReferencedArrivalInterior

`EphemerisReferencedArrivalExterior` derives from `EphemerisReferencedArrival`, and represents the case where the boundary point lies on the edge of a triaxial ellipsoid surrounding the *central body of the current journey's universe*. The spacecraft can then be thought of as *exiting* the ellipsoid from the *interior*. Relevant examples include departing the sphere of influence of a body. `EphemerisReferencedArrivalExterior` is an abstract base class for several boundary classes as defined below.

On the right-hand side of the boundary, the state vector is transformed by *adding* the position and velocity of the central body relative to the *next* journey's central body, thus transforming the state into the frame of the next journey.

13.6.12 EphemerisReferencedLTRendezvousInterior

`EphemerisReferencedLTRendezvousInterior` is the simplest form of `EphemerisReferencedArrivalInterior` in which the spacecraft comes to a rest relative to the ephemeris point at the edge of the bounding ellipsoid. `EphemerisReferencedLTRendezvousInterior` does not include any additional decision variables or constraints.

13.6.13 EphemerisReferencedArrivalWithVinfinityInterior

`EphemerisReferencedArrivalWithVinfinityInterior` extends `EphemerisReferencedArrivalInterior` by adding three decision variables for the magnitude, right ascension, and declination of the velocity vector in the ICRF. `EphemerisReferencedArrivalWithVinfinityInterior` serves as an abstract base class for ephemeris-referenced "exterior" boundary events that require a velocity vector relative to the bounding ellipsoid. The user defines the bounds on the velocity magnitude.

13.6.14 EphemerisReferencedInterceptInterior

`EphemerisReferencedInterceptInterior` is a derived class of `EphemerisReferencedArrivalWithVinfinityInterior` that describes a spacecraft arriving at the bounding ellipsoid with a relative velocity and an optional impulsive TCM. The user defines the size of the TCM and the performance is calculated from the spacecraft's monoprop system as defined in Chapter 5.

13.7 FreePointBoundary

`FreePointBoundary` is a base class for all boundary conditions that begin at a point in space that is defined as a cartesian or classical orbit elements (COE) state relative to the central body. The user may choose to fix or vary within bounds any of the six elements of the position and velocity state on the left-hand side of the boundary. If the user chooses to fix any of these values, they are still variables but their bounds are ±1.0e-13, so the solver cannot move them. EMTG also encodes a mass on the left-hand side of the boundary event. Some derived classes of `FreePointBoundary` may encode additional variables as discussed below.

The user may choose a frame to encode their `FreePointBoundary`. That frame applies both to the bounds/fixed values and also to the initial guess.

13.7.1 FreePointDeparture

`FreePointDeparture` derives from both `FreePointBoundary` and `DepartureEvent`, and is an abstract base class for several boundary classes as defined below.

If a `FreePointDeparture` of any kind occurs at the beginning of a journey after the previous journey ended in a `FreePointArrival` or `EphemerisReferencedArrival`, then the 6-state is not encoded and instead is drawn from the previous boundary event.

13.7.1.1 FreePointFreeDirectDeparture

`FreePointFreeDirectDeparture` is the simplest form of `FreePointDeparture` in which the spacecraft takes on the position and velocity of the free point. Mass is chosen as a decision variable. `FreePointFreeDirectDeparture` has a wait time, and therefore in journeys after than the first, its `DepartureEvent` base class will create a mass continuity constraint. If the `FreePointFreeDirectDeparture` is the first event in the mission, then the user may choose to either fix the mass (by setting `allow_initial_mass_to_vary` to false), or allow the mass to vary between zero and the user-defined `maximum_mass` (by setting `allow_initial_mass_to_vary` to true).

13.7.1.2 FreePointDirectInsertion

`FreePointDirectInsertion` describes an impulsive departure from a free point. Three new decision variables are added - the magnitude of the departure v_∞, and the right ascension and declination of the departure asymptote in the ICRF. The user provides the bounds for all three decision variables.

If the `FreePointDirectInsertion` describes the first event in a mission, then the user may choose to fix the departure mass (by setting `allow_initial_mass_to_vary` to false) or to allow it to vary up to a user-defined `maximum_mass` (by setting `allow_initial_mass_to_vary` to true).

If the `FreePointDirectInsertion` describes a later event in the mission, then a mass continuity constraint is applied to ensure that the mass at departure masses the previous event's mass at arrival.

The departure maneuver is modeled as an impulsive burn using the spacecraft's thrusters as per Section **??**. The user may opt to constrain the departure maneuver to be along the boundary point's velocity vector by setting the `force_free_point_direct_insertion_along_velocity_vector` flag.

13.7.2 FreePointArrival

`FreePointArrival` derives from both `FreePointBoundary` and `ArrivalEvent`, and is an abstract base class for several boundary classes as defined below.

13.7.2.1 FreePointLTRendezvous

`FreePointLTRendezvous` is the simplest form of `FreePointArrival` and represents matching position and velocity with the free point. No additional variables or constraints are added.

13.7.2.2 FreePointArrivalWithVinfinity

`FreePointArrivalWithVinfinity` is an abstract base class for two derived classes below. It adds three new decision variables for the three components of $\mathbf{v}_\infty$ and a user-defined constraint on the magnitude, v_∞. The bounds on the $\mathbf{v}_\infty$ components are restricted to be $\pm$ the upper bound on v_∞.

13.7.2.3 FreePointIntercept

`FreePointIntercept` is a derived class of `FreePointArrivalWithVinfinity` that does not add any new capabilities except to be not abstract. This represents the scenario where the spacecraft has to match position with the free point but not velocity. `FreePointIntercept` can perform the mass drop and propellant consumption associated with a fixed-magnitude TCM. If applied, the TCM is done on the spacecraft's monoprop system as defined in Chapter 5.

13.7.2.4 FreePointChemRendezvous

`FreePointChemRendezvous` is a derived class of `FreePointArrivalWithVinfinity` that adds a maneuver to match velocity with the free point. This maneuver is be performed on the spacecraft's biprop system as defined in Chapter 5. `FreePointChemRendezvous` can perform the mass drop and propellant consumption associated with a fixed-magnitude TCM. If applied, the TCM is done on the spacecraft's monoprop system as defined in Chapter 5.

13.8 PeriapseBoundary

`PeriapseBoundary` is an abstract base class for all boundary events that happen at periapse of the spacecraft's orbit about the central body. In the current implementation, PeriapseBoundary only guarantees that the spacecraft be at *an* apse, not necessarily the right one. In practice this has never been a concern, but we could fix it some day. Note that if a state representation that includes true anomaly (COE, IncomingBplane, or OutgoingBplane) is chosen, then a periapse is guaranteed.

`PeriapseBoundary` is a wrapper on top of `FreePointBoundary`. Unlike in `FreePointBoundary`, the user does not set bounds on each state variable. Rather, these are computed automatically.

`PeriapseBoundary` automatically imposes two constraints:

1. If the chosen state representation does not directly encode distance (`SphericalAZFPA` and `SphericalRADEC` do this), then a distance constraint is imposed. The bounds for the distance constraints are drawn from the user-specified arrival or departure altitude bounds, as appropriate.
2. If the chosen state representation does not directly encode true anomaly or flight path angle, then a constraint is imposed to guarantee $\mathbf{r} \bullet \mathbf{v} = 0$.

13.8.1 PeriapseDeparture

`PeriapseDeparture` derives from both `PeriapseBoundary` and `DepartureEvent`, and is an abstract base class for several boundary classes as defined below.

If the user has defined `IncomingBplane` as the periapse boundary state representation, then `PeriapseBoundary` will switch it to `OutgoingBplane` for the purpose of this departure event only.

13.8.2 PeriapseLaunchOrImpulsiveDeparture

`PeriapseLaunchOrImpulsiveDeparture` represents the case of a spacecraft departing from periapse of an orbit about the central body by means of a launch model. `PeriapseLaunchOrImpulsiveDeparture` *always* uses the `OutgoingBplane` state representation regardless of the user's choice of state representation. The bounds on the boundary event's **DHA!** (**DHA!**) are set to conform to the user-specified bounds on DLA. Also, `PeriapseLaunchOrImpulsiveDeparture` is only permitted to be the first event of the mission.

The C_3 and passed to the launch vehicle code (Section 5.2) to determine the maximum allowable mass. The encoded mass of the vehicle is then either constrained to match the launch vehicle capability (if `allow_initial_mass_to_vary` is false), or to be less than or equal to the launch vehicle capability (if `allow_initial_mass_to_vary` is true).

This boundary event is most commonly used to describe, in reasonably high fidelity, the departure of a spacecraft from a parking orbit during launch. The right ascension (RA) and declination (DEC) entries in the .emtg summary line for `PeriapseLaunchOrImpulsiveDeparture` represent the Right Ascension of Launch Asymptote (RLA) and DLA, *not* the RA and DEC of the departure impulse.

13.8.3 PeriapseArrival

`PeriapseArrival` derives from both `PeriapseBoundary` and `ArrivalEvent`, and is an abstract base class for several boundary classes as defined below.

If the user has defined `OutgoingBplane` as the periapse boundary state representation, then `PeriapseBoundary` will switch it to `IncomingBplane` for the purpose of this departure event only.

13.8.4 PeriapseFlybyIn

`PeriapseFlybyIn` represents the case where the spacecraft arrives at periapse of an orbit relative to the central body and matches position and velocity. The user defines the bounds on the magnitude of the position vector (*i.e.* the radius value). All other computations are handled by the `PeriapseArrival` and `PeriapseBoundary` base classes. This boundary event is typically used to represent periapse of a gravity assist maneuver, but when combined with orbit element constraints as described in Section 13.9, can also be used to model orbit insertion.

13.9 Boundary Constraints

At construction, each boundary event object constructs a vector of constraint objects, all of which inherit from `SpecializedBoundaryConstraintBase`.

The boundary events are constructed by the `SpecializedBoundaryConstraintFactory()` function. At run-time, `BoundaryEventBase` calls `calcbounds()`, `process_constraint()`, and `output()` on each constraint object.

The boundary event constraints are listed in full, both in terms of design and also user documentation, in the scripted constraints document. The individual constraints are not listed here in the software design document because we wanted to keep all of the scripted constraints together in one place. EMTGv9's constraint architecture is designed such that a developer, or even a user, can write a new constraint without touching the rest of the program.

13.9.1 Orbit Element Constraints

A subset of the available specialized boundary constraints are specified with respect to the classical orbit elements at the boundary point, in a frame of the user's choice. All such constraints inherit from the base `OrbitElementConstraintBase` class, itself a derived class of `SpecializedBoundaryConstraintBase`.

Donald will describe how the base orbit element constraint class interacts with `BoundaryEventBase` to retrieve the orbit elements and their derivatives.

Chapter 14

Objective Functions

14.1 Overview

EMTG provides a variety of different objective functions that the user may use in an optimization problem. Objective functions compute their own bounds and sparsity patterns, and interact with the rest of the EMTG problem via a pointer to the mission object. All objective functions inherit from `ObjectiveFunctionBase` and are constructed by `ObjectiveFunctionFactory`. EMTG's optimizer is always configured to make the objective function as small as possible. Accordingly, any objective functions that seek to maximize a quantity do so by minimizing the negative of that quantity.

All objective functions contain the following methods:

- `calcbounds()` - Calculates the sparsity pattern of the objective function.
- `process()` - Computes the value of the objective function and its partial derivatives.
- `output()` - Writes the name and value of the objective function to the .emtg summary file.

14.2 MinimizeDeltavObjective

`MinimizeDeltavObjective` minimizes the total deterministic Δv in the mission, including any Δv performed during a boundary event. `MinimizeDeltavObjective` relies on each phase's `process_deltav_contribution()` method to work. Since process_deltav_contribution() is only fully implemented in MGAnDSMs, MinimizeDeltavObjective only works with missions that are entirely composed of MGAnDSMs phases. This turns out not to be a significant handicap because Δv is a poor objective function for low-thrust missions anyway because it does not map directly to propellant consumption or any other spacecraft system metric.

14.3 MaximizeMassObjective

`MaximizeMassObjective` maximizes the final mass of the spacecraft at the end of the mission, including any propellant margin and additional mass that may have been added during the mission such as samples from a small body. `MaximizeMassObjective` operates only on the mass element of the state vector in the final phase's arrival event. This is usually directly encoded in the decision vector, but `MaximizeMassObjective` does not explicitly require this.

14.4 MaximizeLogeMassObjective

`MaximizeLogeMassObjective` is the natural log of `MaximizeMassObjective`. This is sometimes better behaved numerically, especially on MGAnDSMs problems.

14.5 MaximizeLog10MassObjective

`MaximizeLog10MassObjective` is the log base 10 of `MaximizeMassObjective`. This is sometimes better behaved numerically than `MaximizeMassObjective`, especially on MGAnDSMs problems.

14.6 MaximizeDryMassObjective

`MaximizeDryMassObjective` is similar to `MaximizeMassObjective` except that propellant margin is removed. This is done by adding up the virtual tank variables for all phases and boundary events associated with the final stage of the spacecraft, then scaling each tank value by its user-supplied margin factor. Sometimes optimizing on `MaximizeDryMassObjective` gives a different answer than `MaximizeMassObjective` because if you have to spend more propellant to deliver more final mass, you may have to give up more of that final mass in propellant margin.

14.7 MaximizeLog10DryMassObjective

`MaximizeLog10DryMassObjective` is the natural log of `MaximizeDryMassObjective`. This is sometimes better behaved numerically, especially on MGAnDSMs problems.

14.8 MaximizeLogeDryMassObjective

`MaximizeLogeDryMassObjective` is the natural log of `MaximizeDryMassObjective`. This is sometimes better behaved numerically, especially on MGAnDSMs problems.

14.9 MinimizeTimeObjective

`MinimizeTimeObjective` minimizes the total mission flight time by summing each phase flight time, wait time, and finite boundary event time width (for spiral segments). `MinimizeTimeObjective` is posed to the optimizer as a linear constraint.

14.10 MaximizeInitialMassObjective

`MaximizeInitialMassObjective` maximizes the initial mass of the spacecraft. It operates only on the mass element of the state vector in the first phase's departure event.

14.11 ArriveAsEarlyAsPossibleObjective

`ArriveAsEarlyAsPossibleObjective` minimizes the epoch of the final event in the mission. This is not the same thing as minimizing flight time, as it will move the launch date within user-specified bounds as needed in order to arrive as early as possible.

14.12 ArriveAsLateAsPossibleObjective

`ArriveAsLateAsPossibleObjective` maximizes the epoch of the final event in the mission.

14.13 DepartAsEarlyAsPossibleObjective

`DepartAsEarlyAsPossibleObjective` minimizes the epoch of the first event in the mission. This epoch is usually encoded directly in the decision vector, but `DepartAsEarlyAsPossibleObjective` does not explicitly require it.

14.14 DepartAsLateAsPossibleObjective

`DepartAsEarlyAsPossibleObjective` maximizes the epoch of the first event in the mission. This epoch is usually encoded directly in the decision vector, but `DepartAsEarlyAsPossibleObjective` does not explicitly require it.

14.15 MinimizeChemicalFuelObjective

`MinimizeChemicalFuelObjective` minimizes the total consumption of chemical fuel across the mission. It does so by summing all virtual chemical fuel variables in the mission, regardless of what spacecraft stage they are attached to. Note that this is not the same thing as minimizing the sum of chemical fuel and oxidizer. This objective function is very useful if the spacecraft's oxidizer tank is constrained and the user wishes to minimize the amount of fuel used in both monoprop and biprop maneuvers while strictly obeying the oxidizer constraint.

14.16 MinimizeElectricPropellantObjective

`MinimizeElectricPropellantObjective` minimizes the use of electric propellant in the mission. It does so by summing all virtual electric propellant variables in the mission, regardless of what spacecraft stage they are attached to.

14.17 MinimizeTotalPropellantObjective

`MinimizeTotalPropellantObjective` minimizes the sum of all types of propellant in the mission by summing all virtual propellant variables.

14.18 MinimizeWaypointTrackingErrorObjective

`MinimizeWaypointTrackingErrorObjective` minimizes the average distance between the spacecraft and a user-supplied reference trajectory. This objective function is not yet fully implemented and when complete will be compatible only with parallel shooting phase types. The distance between the spacecraft and the reference trajectory will be sampled at the left-hand side of each parallel shooting step. Distance will be calculated in terms of either Euclidean distance or Mahalanobis distance.

14.19 MinimizeInitialImpulseObjective

`MinimizeInitialImpulseObjective` minimizes the initial impulse in the first phase's departure event. This objective function is most commonly used to minimize launch C_3.

14.20 MaximizeDistanceFromCentralBodyObjective

`MaximizeDistanceFromCentralBodyObjective` maximizes the distance between the spacecraft and the final journey's central body at the end of the mission. This objective function is commonly used in planetary defense applications after an `EphemerisPeggedMomentumTransfer` as described in Section 13.5.2.7. The remainder of the mission after the `EphemerisPeggedMomentumTransfer` represents the path of a solar system body that has been deflected in some way, and the final journey's universe is centered on the Earth. `MaximizeDistanceFromCentralBodyObjective` therefore maximizes the distance by which the body misses the Earth.

Chapter 15

Solvers

15.1 Overview

EMTG contains both a local optimizer and a stochastic global search heuristic. The user may choose to run the local optimizer and the stochastic search heuristic, just the local optimizer, or no optimizer at all and instead evaluate an initial guess with no iterations.

15.2 Gradient-Based Solver

EMTG formulates its optimization problems as NLP problems. The optimizer solves a problem of the form:

$$\begin{aligned} &\text{Minimize } f(\mathbf{x}) \\ &\text{Subject to:} \\ &\mathbf{x}_{lb} \leq \mathbf{x} \leq \mathbf{x}_{ub} \\ &\mathbf{c}(\mathbf{x}) \leq \mathbf{0} \\ &A\mathbf{x} \leq \mathbf{0} \end{aligned} \tag{15.1}$$

where $\mathbf{x}_{lb}$ and $\mathbf{x}_{ub}$ are the lower and upper bounds on the decision vector, $\mathbf{c}(\mathbf{x})$ is a vector of nonlinear constraint functions, and A is a matrix describing any linear constraints (*e.g.* time constraints).

Most interplanetary trajectory optimization problems consist of hundreds of variables and tens to hundreds of constraints. Such problems are best solved with a *sparse* NLP solver such as Sparse Nonlinear OPTimizer (SNOPT) [10]. SNOPT uses a sparse sequential quadratic programming (SQP) method and benefits greatly from precise knowledge of the problem Jacobian, *i.e.*, the matrix of partial derivatives of the objective function and constraints with respect to the decision variables. EMTG provides analytical expressions for all of the necessary partial derivatives, leading to improved convergence *vs.* using numerically approximated derivatives [6, 7, 11]. SNOPT, like all NLP solvers, requires an initial guess of the solution and tends to converge to a solution in the neighborhood of that initial guess. The next section discusses EMTG's fully automated method

for generating initial guesses.

EMTG's NLP solver interface consists of the `NLP_interface` abstract base class and the `NLPoptions` data structure. Individual solvers are addressed via derived classes of `NLP_interface`. Currently the only such derived class is `SNOPT_interface`. EMTG could interface to other NLP solvers such as IPOPT, SOS, WORHP, *etc.* if the need arose and if licenses to those solvers were provided.

15.3 MBH

15.3.1 Monotonic Basin Hopping

EMTG has the ability to search for globally optimal solutions and to optimize without an initial guess via the MBH stochastic global search heuristic [12–23].

MBH [24] is an algorithm for searching for the best solutions to problems with many local optima. Many problems, including those described in this work, are structured such that individual locally optimal "basins" cluster together, where the distance in the decision space from one local optima to the next in a given cluster may be traversed in a short "hop." A problem may have several such clusters. MBH was originally developed to solve molecular conformation problems in computational chemistry, but has been demonstrated to be effective on various types of interplanetary trajectory problems [12, 18–20, 25, 26]. Pseudocode for MBH is given in Algorithm 1, and a diagram of the MBH process on a 1-dimensional function is shown in Figure 15.1.

Special attention is given to decision variables that define the time-of-flight between two boundary points, *e.g.* Earth or Trojan flybys in Lucy. These are the most significant variables that define at trajectory and therefore it is sometimes necessary to drastically perturb them in order to "hop" to a new cluster of solutions. With some (low) uniform-random probability ρ, each time-of-flight variables is shifted by $\pm$ 1 synodic period of the two boundary points defining that trajectory phase. In preliminary design for Lucy, ρ was set to 0.05. In high fidelity re-optimization, ρ is set to 0.0, because we do not expect significant changes to the trajectory.

MBH is run until either a specified number of iterations (trial points attempted) or a maximum CPU time is reached, at which point the best solution stored in the archive is returned. The version of MBH used in EMTG has two parameters: the stopping criterion and the type of random step used to generate the perturbed decision vector $\mathbf{x}'$. In this work, the random step is drawn from a bi-directional Pareto distribution with the Pareto parameter, α, set to 1.4. The bi-directional Pareto distribution usually generates small steps that allow MBH to *exploit* the local cluster around the current best solution. However, some of the steps generated by the bi-directional Pareto distribution are much larger, in some cases spanning the entire decision space. These larger steps allow MBH to *explore* the full decision space. This approach has been shown to be robust on complex interplanetary trajectory design problems [22].

MBH may be started either from a uniform-randomly chosen point in the decision space or from an initial guess derived from a previous problem. As long as the two problems are sufficiently similar, the latter approach is more efficient than starting from randomness. However, on simple

Algorithm 1 Monotonic Basin Hopping (MBH)

generate random point $\mathbf{x}$
run NLP solver to find point $\mathbf{x}^*$ using initial guess $\mathbf{x}$
$\mathbf{x}_{current} = \mathbf{x}^*$
if $\mathbf{x}^*$ is a feasible point **then**
    save $\mathbf{x}^*$ to archive
while not hit stop criterion **do**
    generate $\mathbf{x}'$ by randomly perturbing $\mathbf{x}_{current}$
    for each time-of-flight variable t_i in $\mathbf{x}'$ **do**
        if $rand\,(0,1) < \rho_{time-hop}$ **then**
            shift t_i forward or backward one synodic period
    run NLP solver to find locally optimal point $\mathbf{x}^*$ using in initial guess $\mathbf{x}'$
    if $\mathbf{x}^*$ is feasible **and** $f\,(\mathbf{x}^*) < f\,(\mathbf{x}_{current})$ **then**
        $\mathbf{x}_{current} = \mathbf{x}^*$
        save $\mathbf{x}^*$ to archive
    else if $\mathbf{x}^*$ is infeasible **and** $\|c\,(\mathbf{x}^*)\| < \|c\,(\mathbf{x}_{current})\|$)
        $\mathbf{x}_{current} = \mathbf{x}^*$
return best $\mathbf{x}^*$ in archive

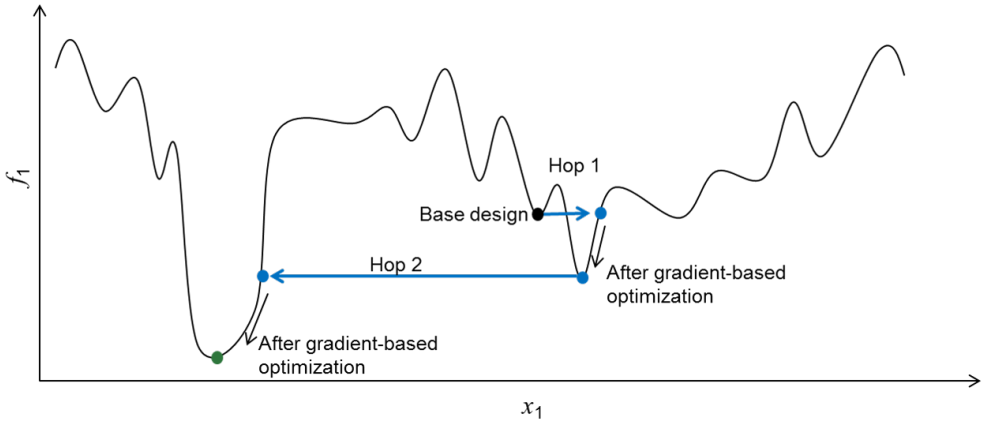


Figure 15.1: Monotonic Basin Hopping on a 1-dimensional function.

problems, MBH will quickly find the global best solution with or without an initial guess.

15.4 Initial Guess

If the user chooses to specify an initial guess to MBH or to run EMTG in NLP-only or evaluate-only mode, then an initial guess must be provided. EMTG features an intelligent initial guess parser that is designed to minimize load on the user. This process is actually executed by the `Mission` class but is described here because this is where users would expect to find it. The initial guess parser obeys the following rules:

1. The initial guess is specified at the Journey level, in the `trialX` field of each `JourneyOptions` object. The user may type or paste the initial guess into the corresponding `trialX` field for each journey in the .emtgopt input script.
2. Each entry in the initial guess is a comma-separated tuple. The first element is the name of the decision variable and the second element is the value. The name of each decision variable can be found by running EMTG with no initial guess and reading the resulting XFfile.csv that lists all decision variables and constraints.
3. While EMTG's internal time calculations are performed in seconds, we recognize that humans like to think in days and so time variables in the initial guess are parsed in days.
4. If any decision variable is *not* specified, EMTG will assign a uniform random number between the bounds of that variable. It is therefore possible to supply a partial initial guess. EMTG warns the user when it does this.
5. Decision variables related to a number of time steps, such as control variables in a `TwoPointShootingPhase` or state and control variables in a `ParallelShootingPhase` are interpolated to the number of time steps present in the phase. The user may specify an initial guess with a different number of time steps than the mission is configured for - EMTG will figure it out.
6. Decision variables in the initial guess whose names are not recognized are ignored.

Chapter 16

Math

16.1 Matrix

EMTG performs all matrix and vector computations using the `EMTG_Matrix` class. `EMTG_Matrix` provides dense matrix functionality that is templated and can be used with regular double precision numbers, integers, booleans, or algorithmic differentiation calculation objects. For a full exhaustive list of methods available in `EMTG_Matrix`, see the Doxygen output.

The developer may turn matrix error checking on and off at compile time. It is very useful to have this checking turned on during development but once code is fully developed and tested the error checking is removed for speed.

16.2 Tensor

`EMTG_Tensor` is similar to `EMTG_Matrix` but for 3-tensors. `EMTG_Tensor` implements fewer operations than `EMTG_Matrix` because tensors are very rare in EMTG and not many operations are needed. A full list of methods in `EMTG_Tensor` is included in the Doxygen output.

16.3 Interpolator

The `interpolator` class is a very simple linear interpolator that is used by EMTG when parsing initial guesses. It operates on a `std::vector` of `std::pair` objects.

16.4 EMTG math utilities and constants

The `EMTG_math.h` header contains mathematical constants and basic functions that do not fit anywhere else. These include:

- Definitions of π, $\pi/2$, and 2π.
- The conversion from degrees to radians.
- A definition of a "small" number in EMTG, currently set to 1.0e-13.
- A definition of a "large" number in EMTG, currently set to 1.0e+30.
- An `sgn()` function to find the sign of a value. Works for double, int, and algorithmic differenation overloaded computation objects.
- `acosh()` and `asinh()` functions, compatible with algorithmic differentiation.
- `safe_acos()` and `safe_asin()` functions that clip the input argument between 0.0 and 1.0 and so never fail, compatible with algorithmic differentation.
- A `norm()` function for `std::vector`, which is rarely used but sometimes helpful. Compatible with algorithmic differentation.
- An `absclip()` function to return a value clipped between plus or minus a user-supplied maximum absolute value. Compatible with algorithmic differentiation.

16.5 RandUtils

EMTG uses the `randutils` package by Melissa O'Neill to generate random numbers. This is the only third-party code that is distributed with the EMTG code base, and is released under the MIT License. The license is included in its entirety at the beginning of `randutils.h`.

Chapter 17

Derivative Testbed

One of the most challenging aspects of creating an optimization tool is the need to code, test, and maintain analytical derivatives of each of the objective functions and constraints with respect to the decision variables. This testing is often done with finite differencing or by creating stand-alone testbeds that use algorithmic differentiation or complex step differentiation via operator overloading. Neither of these methods is reliable because the former is subject to finite differencing rounding and truncation error and the latter introduces a risk of human error when re-coding a derivative from the stand-alone testbed to the main program code.

After many frustrating years of using both of the above methods, we decided to build EMTG version 9 a very different way. The entire EMTG version 9 code base is compatible with algorithmic differentation via the GSAD package. All values that are used in the computation of a constraint or objective function are specified as `doubleType`, which in turn may be cast as `double` or `GSAD::adouble` depending on whether one is compiling in optimization mode or testbed mode.

The derivatives testbed executes EMTG in "evaluate trialX" mode with algorithmic differentation available. The testbed prints out the following files:

- `Mission_XFout.csv` - Lists the decision variables and constraints along with their bounds and values. This is the same file that is written when executing EMTG the normal way.
- `Mission_Gout.csv` - Lists every single partial derivative of every nonlinear constraint and the objective function with respect to every decision variable in its sparsity pattern. Lists the analytical value, the algorithmic differentiation value, the relative and absolute errors, the ratios between the analytical and algorithmic values, the indices in the sparse Jacobian, and the order of magnitude of the derivative entry.
- `Mission_Aout.csv` - Same as `Mission_Gout.csv` but for linear constraints and objectives.
- `Mission_MissingEntries.csv` - Lists the Jacobian indices and order of magnitude of any derivative entry that *should* exist based on the derivative directions polled from each algorithmic differentation calculation object, but the programmer forgot to code.

The derivatives testbed allows a developer to trivially test a highly complex optimization problem and ensure that the derivatives are as correct as possible. The only complications are when an entry is so small that the true value is obscured by floating point chaos.

Developers may also construct smaller-scale derivatives testbeds that test only portions of EMTG. The repository includes testbeds for boundary events, propagation, the MGALT STM-MTM chain, and frame rotations. These secondary testbeds are not maintained like the main testbed and are provided only as examples of how to make tests.

Chapter 18

Regression Testbed

Alec and Sean

The `Regression Testbed` for EMTG is, by default, located in the "testatron/" folder in the EMTG installation. All test cases are located in subfolders within the "testatron/tests/" directory. [SEAN, MAYBE SUMMARIZE TESTS HERE?] Each test case includes an EMTG options (.emtgopt) file for the `Regression Testbed` to rerun as well as a baseline output (.emtg) file to use as a truth value for test runs. All required universe and ephemeris files are located in the "testatron/universe/" folder. All required launch vehicle (.emtg_launchvehicleopt), power system (.emtg_powersystemsopt), propulsion system (.emtg_propulsionsystemopt), and spacecraft (.emtg_spacecraftopt) options files for the test cases are located in the "testatron/HardwareModels/" folder along with all throttle (.Throttle), throttle table (.ThrottleTable), and throttle table output (.ThrottleTableOUTPUT) files. The `Regression Testbed` uses `Testatron` (outlined in 18.1) to run and compare EMTG cases. `Testatron` uses the `Comparatron` or `Comparatron_NoCoast` methods (outlined in 18.2) to compare Mission, Journey, and Mission Event attributes between the new EMTG run created by `Testatron` and the baseline output provided in the "testatron/tests/" folder for each test case.

18.1 Testatron

`Testatron` is the driver for the EMTG regression testing system and is located in the "testatron.py" python script within the "testatron/" folder. `Testatron` searches through subdirectories in the "testatron/tests/" folder for EMTG options files to be run and compared—all subfolders appearing in the "testatron/tests/" folder can be used or, alternatively, a specific set of test subfolders or cases can be provided as a list in the `Testatron` script. For each EMTG options file found, the driver will override the output path to be a time stamped output folder in your current working directory. It also makes sure that background mode is on and that the universe and HardwareModels paths are set to those in the "testatron" folder. `Testatron` then saves and runs the EMTG options file and puts the newly-generated output through a comparator (outlined in 18.2), to be validated against a baseline case.

There are multiple ways to run `Testatron`, all of which can be specified by the "run_type", "test_cases", and "skip_coasts" arguments at the top of the script. The "run_type" argument has four options:

- *all*: Runs all test cases that appear in any subfolder in the tests directory
- $folders$: Runs all test cases in user-specified subfolders in the tests directory
- *cases*: Runs user-specified test cases
- $failed$: Runs only the cases that failed in a user-specified previous `Testatron` run or runs

For the $folders$, *cases*, and $failed$ options, the user must provide additional information into the "test_cases" argument in order for `Testatron` to know what to run. For the $folders$ option, a list of subfolders within the "testatron/tests/" directory must be provided. For the *cases* option, a list of EMTG test case names (including full file path) must be provided. For the $failed$ option, the full file path to the `Testatron` output folder (or folders) must be provided. If `Testatron` is being run using the *all* option, then the "test_cases" argument can be ignored. Lastly, the "skip_coasts" argument controls which version of `Comparatron` will be used. If "skip_coasts" is set to False, the standard `Comparatron` method will be used and if "skip_coasts" is set to a boolean value of True, then the `Comparatron_NoCoast` method will be used.

As all the test cases are run, two comma-separated value files are written: "test_results.csv" and "failed_tests.csv". The "test_results.csv" file is an overall summary file that provides a time stamp for the beginning and end of the tests as well as the successful/failed status for each individual test. The "failed_tests.csv" file will only be written to if one of the tests fails in the `Comparatron`. This CSV file provides the name, values, error, and tolerance for every Mission, Journey, or Mission Event attribute that failed.

18.2 Comparatron and Comparatron_NoCoast Methods

`Comparatron` is a method of the Mission class (located in "PyEMTG/Mission.py") which compares all EMTG output attributes against those of a baseline case. Strings are compared directly and numeric values are checked against a default tolerance of 1e-15. Alternative tolerance values for any attribute can be provided as a dictionary into the function. If all values are in agreement, `Comparatron` returns a boolean value True. If any values are not within the tolerance between the two cases, then a value of "False" is returned along with a dataframe summarizing all inconsistencies.

The syntax for calling `Comparatron` is: "$pass_test = myMission.Comparatron(path_to_baseline, csv_file_name = None, full_output = False, tolerance_dict = \{\}, default_tolerance =$1e-15)." Note that this is Python syntax. Any argument with an "=" sign denotes the default value that will be used if the argument is not passed in. The function arguments are as follows:

- *path_to_baseline*: The full file path to the EMTG output file to compare against. This is the only required argument.

- *csv_file_name*: The name of the csv output file that is written if there are any discrepancies to report. If no file name is provided then the mission name will be used.
- *full_output*: Dictates what is written to the csv output file. If False, only values that do not meet the tolerance will be written to the output. If True, than any values that are not in exact agreement (regardless of whether they are within the tolerance) will be written.
- *tolerance_dict*: Overrides the default tolerance for specific mission, journey, or mission event attributes. The argument takes a dictionary with the attribute as the key and the new tolerance as the value (i.e. {'total_statistical_deltav':1e-6,'Declination':1e-8}).
- *default_tolerance*: Provides a default tolerance value for all attributes that do not appear in the tolerance dictionary. A value of 1e-15 will be used unless the user provides a new default value here.

When running `Comparatron`, the function will first check to make sure that both cases have the same number of journeys and mission events. If there are discrepancies in either, the function will not return the usual True/False boolean but will instead return a string saying "Journey Mismatch" or "Journey # Mission Events Mismatch" and will stop immediately. If this check is passed, `Comparatron` will then check every Mission, Journey, and MissionEvent class attribute across cases and record any names that do not agree or any attributes that appear only in one—these will be written to the csv output file. The attributes for each class include both functions within the class as well as alphanumeric values (i.e. dates, the central body, and Δv). All alphanumeric-valued attributes are parsed into a temporary dataframe for the overall mission and for each journey and mission event. These temporary dataframes are split in two, one for strings and one for numerics. The strings are compared directly while the corresponding numeric dataframes between the two EMTG cases are subtracted and checked against the tolerance. All values that do not completely agree are stored into a final comparison dataframe, which gets written to a csv output file if discrepancies are found.

`Comparatron_NoCoast` is also a method of the Mission class and is identical to `Comparatron` in almost every way. The difference between `Comparatron_NoCoast` and `Comparatron` is that `Comparatron_NoCoast` ignores all Mission Events with a "coast" event type. This provides a faster comparator option, but no longer compares every line between two EMTG output files since some Mission Events are being skipped.

Bibliography

[1] B. A. Archinal, C. H. Acton, M. F. A'Hearn, A. Conrad, G. J. Consolmagno, T. Duxbury, D. Hestroffer, J. L. Hilton, R. L. Kirk, S. A. Klioner, D. McCarthy, K. Meech, J. Oberst, J. Ping, P. K. Seidelmann, D. J. Tholen, P. C. Thomas, and I. P. Williams, "Report of the IAU Working Group on Cartographic Coordinates and Rotational Elements: 2015," *Celestial Mechanics and Dynamical Astronomy*, Vol. 130, Feb 2018, p. 22, 10.1007/s10569-017-9805-5.

[2] "SPICE Ephemeris," `http://naif.jpl.nasa.gov/naif/`, accessed 6/26/2016.

[3] J. Knittel, J. Englander, M. Ozimek, J. Atchison, and J. Gould, "Improved Propulsion Modeling for Low-Thrust Trajectory Optimization," *AAS/AIAA Space Flight Mechanics Conference, San Antonio, TX*, February 2017.

[4] J. Englander, M. Vavrina, and D. Hinckley, "Global Optimization of Low-Thrust Interplanetary Trajectories Subject to Operational Constraints," *AAS/AIAA Astrodynamics Conference, Napa, Ca*, February 2016.

[5] "NASA's Evolutionary Xenon Thruster (NEXT): Ion Propulsion GFE Component Information Summary for Discovery Missions, July 2014," `http://discovery.larc.nasa.gov/discovery/pdf_files/20-NEXT-C_AO_Guidebook_11July14.pdf`, July 2014.

[6] D. H. Ellison, B. A. Conway, J. A. Englander, and M. T. Ozimek, "Analytic Gradient Computation for Bounded-Impulse Trajectory Models Using Two-Sided Shooting," *Journal of Guidance, Control, and Dynamics*, Vol. 41, No. 7, 2018, pp. 1449–1462, 10.2514/1.G003077.

[7] D. H. Ellison, B. A. Conway, J. A. Englander, and M. T. Ozimek, "Application and Analysis of Bounded-Impulse Trajectory Models with Analytic Gradients," *Journal of Guidance, Control, and Dynamics*, Vol. 41, No. 8, 2018, pp. 1700–1714, 10.2514/1.G003078.

[8] P. J. Enright and B. A. Conway, "Optimal finite-thrust spacecraft trajectories using collocation and nonlinear programming," *Journal of Guidance, Control, and Dynamics*, Vol. 14, No. 5, 1991, pp. 981 – 985.

[9] J. A. Sims and S. N. Flanagan, "Preliminary Design of Low-Thrust Interplanetary Missions," *AAS/AIAA Astrodynamics Specialist Conference*, Girdwood, Alaska, Paper AAS 99-338, August 1999.

[10] P. E. Gill, W. Murray, and M. A. Saunders, "SNOPT: An SQP Algorithm for Large-Scale Constrained Optimization," *SIAM J. Optim.*, Vol. 12, jan 2002, pp. 979–1006, 10.1137/s1052623499350013.

[11] D. H. Ellison, *Robust Preliminary Design for Multiple Gravity Assist Spacecraft Trajectories*. PhD thesis, May 2018.

[12] C. H. Yam, D. D. Lorenzo, and D. Izzo, "Low-thrust trajectory design as a constrained global optimization problem," Vol. 225, SAGE Publications, aug 2011, pp. 1243–1251, 10.1177/0954410011401686.

[13] G. A. Rauwolf and V. L. Coverstone-Carroll, "Near-optimal low-thrust orbit transfers generated by a genetic algorithm," *Journal of Spacecraft and Rockets*, Vol. 33, nov 1996, pp. 859–862, 10.2514/3.26850.

[14] V. Coverstone-Carroll, "Near-Optimal Low-Thrust Trajectories via Micro-Genetic Algorithms," *Journal of Guidance, Control, and Dynamics*, Vol. 20, jan 1997, pp. 196–198, 10.2514/2.4020.

[15] V. Coverstone-Carroll, J. Hartmann, and W. Mason, "Optimal multi-objective low-thrust spacecraft trajectories," *Computer Methods in Applied Mechanics and Engineering*, Vol. 186, jun 2000, pp. 387–402, 10.1016/s0045-7825(99)00393-x.

[16] M. Vavrina and K. Howell, "Global Low-Thrust Trajectory Optimization Through Hybridization of a Genetic Algorithm and a Direct Method," *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, AIAA, aug 2008, 10.2514/6.2008-6614.

[17] J. A. Englander, B. A. Conway, and T. Williams, "Automated Mission Planning via Evolutionary Algorithms," *Journal of Guidance, Control, and Dynamics*, Vol. 35, nov 2012, pp. 1878–1887, 10.2514/1.54101.

[18] J. A. Englander, B. A. Conway, and T. Williams, "Automated Interplanetary Mission Planning," *AAS/AIAA Astrodynamics Specialist Conference, Minneapolis, MN*, AIAA paper 2012-4517, August 2012, 10.2514/6.2012-4517.

[19] J. A. Englander, *Automated Trajectory Planning for Multiple-Flyby Interplanetary Missions*. PhD thesis, University of Illinois at Urbana-Champaign, April 2013.

[20] D. H. Ellison, J. A. Englander, and B. A. Conway, "Robust Global Optimzation of Low-Thrust, Multiple-Flyby Trajectories," *AAS/AIAA Astrodynamics Specialist Conference, Hilton Head, SC*, AAS paper 13-924, August 2013.

[21] D. H. Ellison, J. A. Englander, M. T. Ozimek, and B. A. Conway, "Analytical Partial Derivative Calculation of the Sims-Flanagan Transcription Match Point Constraints," *AAS/AIAA Space-Flight Mechanics Meeting, Santa Fe, NM*, AAS, January 2014.

[22] J. A. Englander and A. C. Englander, "Tuning Monotonic Basin Hopping: Improving the Efficiency of Stochastic Search as Applied to Low-Thrust Trajectory Optimization," *24th International Symposium on Space Flight Dynamics, Laurel, MD*, ISSFD, May 2014.

[23] M. Vavrina, J. Englander, and D. Ellison, "Global Optimization of N-Manuever, High-Thrust Trajectories Using Direct Multiple Shooting," *AAS/AIAA Space Flight Mechanics Meeting*, February 2016.

[24] R. H. Leary, "Global Optimization on Funneling Landscapes," *Journal of Global Optimization*, Vol. 18, No. 4, 2000, pp. 367–383, 10.1023/A:1026500301312.

[25] B. Addis, A. Cassioli, M. Locatelli, and F. Schoen, "A global optimization method for the design of space trajectories," *Computational Optimization and Applications*, Vol. 48, jun 2009, pp. 635–652, 10.1007/s10589-009-9261-6.

[26] J. Englander, M. A. Vavrina, B. J. Naasz, R. G. Merrill, and M. Qu, "Mars, Phobos, and Deimos Sample Return Enabled by ARRM Alternative Trade Study Spacecraft," *AIAA/AAS Astrodynamics Specialist Conference*, AIAA paper 2014-4354, aug 2014, 10.2514/6.2014-4354.