JSC-61777-models/utils/integration

JSC Engineering Orbital Dynamics Integration Model

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

Package Release JEOD v5.1

Document Revision 3.0
July 2023

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

JSC-61777-models/utils/integration

JSC Engineering Orbital Dynamics Integration Model

Document Revision 3.0
July 2023

David Hammen, Gary Turner

Simulation and Graphics Branch (ER7)
Software, Robotics, and Simulation Division
Engineering Directorate

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas

Executive Summary

The Integration Model forms a component of the utilities suite of models within JEOD v5.1. It is located at models/utils/integration.

Propagating the evolution of a vehicle's translational and/or rotational state over the course of a simulation is an essential part of every space-based Trick simulation. The underlying equations of motion for this state propagation yield second order initial value problems. While analytic solutions do exist for a limited set of such problems, the complex and unpredictable nature of the forces and torques acting on a space vehicle precludes the use of analytic methods for a generic solution to these state propagation problems. Numerical integration techniques must be used to solve the problem.

Performing this numerical integration is the primary goal of this model. Several such numerical integration techniques exist. Deciding which technique is best-suited for the problem at hand is a problem-dependent tradeoff between accuracy and computational cost. The Integration Model thus provides a variety of numerical integration techniques.

Purpose

The Integration Model serves primarily as an interface to the standard integration techniques provided by the ER7 Utilities. The Integration Model also provides two special purpose integration techniques – Gauss-Jackson and the Livermore Solver for Ordinary differential equations (LSODE). These techniques are intended to propagate translational state over long time steps as might be desired for interplanetary vehicles or other situations for which the environmental forces are well known.

LSODE is an adaptive technique which determines its optimal step size. Gauss-Jackson is a conventional fixed step size technique which bootstraps itself from a step consistent with familiar methods such as RK4 to a far larger step by means of a doubling strategy. Both LSODE and Gauss-jackson require some level of configuration, however, default settings are provided which will suffice in most cases.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose and Objectives of the Integration Model

The Integration Model comprises several classes that provide various mechanisms for numerically integrating a changing state over time and provide mechanisms for coordinating this integration across multiple bodies. The model is used in JEOD v5.1to integrate the translational and rotational states of dynamic bodies.

The provided capabilities include:

- Numerically integrating the state described by a 2nd order ODE (e.g. a body state) over time using one of a variety of numerical integration techniques.
- Updating time in a manner consistent with the numerical integration technique used to propagate states,
- Providing an interface to the techniques in ER7 Utilities to leverage services including numerical integration and resource management , and
- Constructing an integrator constructor using the Trick integration structure as a guideline.

1.2 Context within JEOD

The following document is parent to this document:

- *JSC Engineering Orbital Dynamics* [10]

The Integration Model forms a component of the utilities suite of models within JEOD v5.1. It is located at models/utils/integration.

1.3 Document History

Author	Date	Revision	Description
David Hammen	September, 2014	3.0	Updated for ER7 Utilities and JEOD 3.x
Gary Turner	February, 2012	2.0	Re-factor of Integration model for JEOD 2.2; Added Gauss-Jackson method.
David Hammen	September, 2010	1.1	Added cyclomatic complexity
David Hammen	February, 2010	1.0	Initial Version

1.4 Document Organization

This document is formatted in accordance with the NASA Software Engineering Requirements Standard [11].

The document comprises chapters organized as follows:

Chapter 1: Introduction -This introduction describes the objective and purpose of the Integration Model.

Chapter 2: Product Requirements -The requirements chapter describes the requirements on the Integration Model.

Chapter 3: Product Specification -The specification chapter describes the architecture and design of the Integration Model.

Chapter 4: User Guide -The user guide chapter describes how to use the Integration Model.

Chapter 5: Inspections, Tests, and Metrics -The inspections, tests, and metrics describes the procedures and results that demonstrate the satisfaction of the requirements for the Integration Model.

Chapter 2

Product Requirements

Requirement Integration_1: Project Requirements

Requirement:

This model shall meet the JEOD project requirements specified in the JEOD top-level document.

Rationale:

This is a project-wide requirement.

Verification:

Inspection

Requirement Integration_2: ER7 Utilities Framework

Requirement:

The Integration Model shall provide extensions of the ER7 Utilities integration framework that enable it to be used in JEOD.

Rationale:

The ER7 Utilities integration framework was made intentionally generic so that it can be used in a Trick and/or a JEOD context.

Remarks:

The intent of this requirement is to extend the ER7 Utilities concept of an integration group and of a time interface to a JEOD-specific setting.

Verification:

Inspection, test

Requirement Integration_3: ER7 Utilities Techniques

Requirement:
The Integration Model shall provide access to all integration techniques defined in the ER7 Utilities integration module.

Rationale:
The intent of creating the ER7 Utilities was to provide a common framework for integration and to provide integration techniques formerly defined in Trick and in JEOD. Not providing access to all those techniques would be counterproductive.

Verification:
Inspection, test

Requirement Integration_4: Long-Arc Integration

Requirement:
The Integration Model shall provide techniques that accurately and efficiently integrate translational state over long spans of time.

Rationale:
This capability is needed for a number of human and automated simulations.

Verification:
Inspection, test

Requirement Integration_5: Extensibility

Requirement:
The Integration Model architecture shall support extensibility to enable the use of techniques not provided by the ER7 Utilities integration module.

Rationale:
Providing all of the plethora of numerical integration techniques that exist is not practical. However, the architecture should be extensible so as to accommodate techniques not provided.

Verification:
Inspection, test

Requirement Integration_6: Support JEOD ODEs

Requirement:
The Integration Model will provide support for the three classes of integration problems encountered in JEOD:

1. Scalar first order ordinary differential equations,
2. Three-vector second order ordinary differential equations, and

3. The Lie group SO3 as a second order ordinary differential equation.

The support shall be in the form of classes that simplify the use of the broader classes of problems supported by the ER7 Utilities and that automatically allocate and deallocate resources per the Resource Allocation Is Initialization (RAII) scheme widely used across the C++ community.

Rationale:
These integration support constructs simplify the use of the integration elsewhere and reduce the chances of lost resources.

Verification:
Inspection, test

Requirement Integration_7: Multiple States

Requirement:
The Integration Model shall provide the ability to integrate multiple states.

Rationale:
A simulation can involve multiple vehicles, each with its own translational and rotational state.

Verification:
Inspection, test

Requirement Integration_8: Multiple Integrators

Requirement:
The Integration Model shall provide the ability to simultaneously integrate states using disparate integration rates and techniques.

Rationale:
A simulation can involve multiple vehicles, and each vehicle may have its own requirements on the propagation of its state. Particularly in situations where the behaviors are governed by different scales of time (e.g. one vehicle in a geo-synchronous orbit and one in a low-Earth-orbit), the optimal integration rates (and, in some cases, methods) may differ from vehicle to vehicle.

Verification:
Inspection, test

Chapter 3

Product Specification

Propagating the evolution of a vehicle's translational and/or rotational state over the course of a simulation is an essential part of every space-based Trick simulation. JEOD assumes a Newtonian universe. The underlying equations of motion that describe the changes in state over time are the second order ordinary differential equations (ODE) given by Newton's second law of motion. The problem addressed by the Integration Model falls into the very broad category of solving such second order initial value problems. While analytic solutions do exist for a limited set of such problems, the complex and unpredictable nature of the forces and torques acting on a vehicle precludes the use of analytic solutions as a generic solution to these state propagation problems. Numerical integration techniques must be used to solve the problem.

Performing this numerical integration is the primary goal of this model. Several such numerical integration techniques exist. Deciding which technique is best-suited for the problem at hand is a problem-dependent tradeoff between accuracy and computational cost. The Integration Model thus provides a variety of numerical integration techniques.

3.1 Conceptual Design

This section describes the key concepts of the Integration Model and of the ER7 Utilities Integration module.

3.1.1 Initial Value Problem

An initial value problem comprises

- A set of variables that describe some state,
- A set of initial values for these state variables,
- A set of ordinary differential equations (ODEs) that describe how state changes with respect to an independent variable (typically time), and
- An interval over which state is to be advanced.

Three initial value problems currently arise in JEOD:

- Propagating the temperature of a radiative surface from one time step to the next. The temperature of the surface varies with time per the Stefan-Boltzmann law.
- Propagating the translational state (position and velocity) of a dynamic body from one time step to the next. The translational state varies with time per Newton's second law.
- Propagating the rotational state (orientation and angular velocity) of a dynamic body from one time step to the next. The rotational state varies with time per the rotational analog of Newton's second law.

The three initial value problems mentioned above in general do not have nice closed-form solutions. Numerical techniques must be used to yield approximations to the solutions to the problems. The principal purpose of the Integration Model is to provide tools that numerically solve such initial value problems. See Shampine[13] for a generic discussion of numerical solution of ordinary differential equations.

3.1.2 Integration Techniques

In the context of the Integration Model, an integration technique is a specific algorithm that approximately propagates state over time per some ODE. There are many different such techniques, several of which are provided by the ER7 Utilities and by JEOD. They range in complexity from the very simple Euler techniques to the rather complex Gauss-Jackson and LSODE integrators. The paragraphs that follow describe various ways to look at different integration techniques.

Does the technique take multiple steps to achieve the desired end point?
All but the very simplest of integration techniques involve multiple steps to reach the desired end point. Most instances of the Runge-Kutta family of integration techniques integrate to intermediate points in the integration interval before making the final step that necessarily hits the end of the integration interval. For example, the midpoint method first integrates to the middle of the integration interval before integrating to the end of the interval, and the classical fourth order Runge-Kutta technique goes to the midpoint twice before going to the end of the interval (which it also hits twice).

Predictor-corrector methods also take multiple steps to reach the desired end point. Each step of a predictor-corrector method advances state to the end of the integration inteval, but these methods move to the end of the interval multiple times. The first step uses a predictor algorithm to predict the value at the end of the interval. The following steps use a different algorithm to correct that value. Simple predictor-corrector algorithms apply the correction but once. More advanced techniques repeatedly apply the corrector until the changes to the corrected state is less than some tolerance.

Does the technique divide the integration interval into subintervals?
This concept is distinct from the above. Integration techniques that do this are adaptive techniques. An adaptive technique uses some underlying integration technique to propagate over a subinterval.

The adaptive part of the technique tries to find an optimal length for the subinterval, neither too long nor too short, that makes the integrator perform close to optimal with regard to accuracy.

Does the technique use historical data?
All but the simplest of predictor-corrector techniques use historical data to improve the accuracy of the prediction/correction. This improved accuracy comes at a cost. One issue is that such techniques are not self-starting. Some other technique must be used to provide the initial set of history data. The other issue is that discontinuities in the derivatives (e.g., a satellite subject to radiation pressure coming out of or going into Earth shadow) can wreak havoc with the historical data. Users of such techniques need to be aware of the pitfalls as well as the advantages of these techniques.

How many derivative evaluations are needed?
One of the key advantages of predictor-corrector techniques over Runge-Kutta techniques is that given techniques of the same order, the former tend to need far fewer evaluations of the derivatives to attain the same level of accuracy if the function to be integrated is analytic. The fewer derivative calls, the better, because evaluating derivatives tends to be very expensive computationally.

What kind of accuracy can be acheived/expected?
This is possibly the most important question of all. Any numerical integration technique is subject to errors inherent to the technique itself. Some techniques are numerically unstable if the time step is too large, and even where they are stable, truncation errors inevitably result from the approximations used by the techniques. These truncation errors tend to decrease as the integration interval decreases, but not indefinitely. At some point, round-off error that is inherent to the use of finite precision arithmetic (e.g., double precision variables) overwhelms the truncation error. For integration intervals smaller than this transition point, decreasing the size of the integration interval increases the error in the solution.

Characterizing the error behavior of the various integration techniques is a key aspect of the testing section of this document.

3.1.3 Trick Integration

The literature on numerical integration of an ODE will typically have the integrator call some function to calculate the derivative. Except for the simplest of integrators, there will be many calls to the derivative function sprinkled throughout the algorithm.

This is not how integration works in a Trick simulation.

In a Trick simulation, derivative class jobs calculate various quantities that eventually result in state derivatives while integration class jobs advance state using those already calculated derivatives. The Trick integration loop calls all of the pertinent derivative class jobs and then calls all of the pertinent integration class jobs.

This simplifies the development of a Trick-based simulation at the expense of making things rather difficult for the developers of an integration technique. Except for the very simplest of integration techniques which call the derivative function once per integration cycle, the myriad calls to the derivative function must be implemented as if the integrator was a reentrant coroutine. The approach taken in all of the ER7 Utilities and JEOD state integration techniques is to emulate this reentrant coroutine behavior via a finite state machine.

3.1.4 ER7 Utilities Integration

The Integration Model largely stood on its own in previous JEOD releases. This is no longer the case in JEOD v5.1. The integration framework and most of the integration techniques have been migrated to the ER7 Utilities package. This section summarizes the key concepts of the ER7 Utilities Integration module.

3.1.4.1 Integrable Object

An integrable object is an object such as a JEOD DynBody that contains time-dependent state information that needs to be propagated over time in accordance with some ordinary differential equation. The integrable object presumably will contain one or more state integrators (see below) to perform this state propagation.

3.1.4.2 State Integrator

A state integrator is an object that uses a specific integration technique to propagate state over time. To conform with the Trick integration scheme, the object's integration function must receive derivatives and finite state as input arguments.

3.1.4.3 Integrator Result

A state integrator returns an integrator result object to indicate whether the integrator successfully achieved the target state and to indicate the amount by which integration time should be advanced.

3.1.4.4 Integrator Result Merger

Oftentimes multiple integrable objects are integrated concurrently. The integrator results returned by the multiple state integrators must be merged to form a single merged integrator result. An integrator result merger object performs these merge operations.

3.1.4.5 Integration Group

An integration group conceptually contains a set of integrable objects that are to be integrated concurrently. All objects in the group have their integrated state advanced at the same dynamic step size. On each step within the integration process, all derivative class jobs associated with the group are called, followed by calls to each of the integration class jobs associated with the group. This means the state integrators used by the objects must be compatible with one another. On each integration step, the state integrators used by the integrable objects in the group must advance state to the same point in time.

3.1.4.6 Integration Controls

An integration controls object works with an integration group to manage the integration process. The group, along with the integrable objects contained by the group, are indifferent to the specific integration technique used to propagate state. Each integration group contains an integration controls object that directs the integration process at the group level to the target point in time. The integration controls object manages the finite state machine that controls the behavior of the state integrators used by the group's integrable objects. The integration controls defers the calling of the integrable objects contained with the integration group to the group. The integration result returned by the group tells the integration controls how to advance the finite state machine and how to advance time.

3.1.4.7 Time

The ER7 Utilities integration module defines time in a very abstract sense. Integration time in Trick is contained within the Trick Integrator object, while time in JEOD is a global set of objects. The only common aspect between these two very disparate concepts is that time is something that needs to be updated as a part of the integration process.

3.1.4.8 Integrator Constructor

There are two key consistency issues regarding the use of multiple state integrators within an integration group. On each integration step, the integration controls' finite state machine directs the state integrators to advance integrable state to some specific point in time. This means that state integrators used within an integration group must behave consistently with regard to the finite state machine managed by the integration controls and with regard to how time is to be advanced.

This presents a potential issue for JEOD because of the global nature of the JEOD time objects and because of the use of multiple state integrators within an integration group. This issue does not arise if the integration controls and state integrators are consistent with one another.

An integrator constructor creates objects that collectively and consistently work together to integrate the states of integrable objects over time. An integrator constructor creates

- State integrator objects that integrable objects can use to integrate state,
- Integrator results merger objects that state integrators and integration groups can use to merge results from multiple integrators, and
- Integration controls objects that integration groups can use to direct the integration process.

Using an integrator constructor ensures that the state integrator objects, integrator results merger objects, and integration controls object used within an integration group are consistent with respect to the finite state machine and with respect to time.

3.1.4.9 Supported Integration Techniques

The ER7 Utilities provides a number of integration techniques. Each integration technique includes a technique-specific integrator constructor that ensures consistency of state integrators and integration controls across an integration group. The paragraphs that follow describe the techniques provided with the ER7 Utilities Integration model.

3.1.4.9.1 Euler's Method

Euler's method is the simplest of integration techniques. It advances the first order ODE $\dot{x}(t) = f(x,t)$ via $x(t+\Delta t) = x(t) + \Delta t\, f(x,t)$. Euler's method is the basis for all of the other integration techniques supplied with the ER7 Utilities Integration model. Euler's method itself exhibits a global error that is first order in time. This means Euler's method typically exhibits very poor accuracy. As a result, its use is not recommended.

See appendix B for details.

3.1.4.9.2 Symplectic Euler Method

The symplectic Euler is the simplest of integration techniques specialized to second order ODE problems. It advances the second order ODE $\dot{x}(t) = v(t)$, $\dot{v}(t) = f(x,v,t)$ via $v(t+\Delta t) = v(t) + \Delta t\, f(x,v,t)$, $x(t+\Delta t) = x(t) + \Delta t\, v(t+\Delta t)$. The symplectic Euler method is only marginally better than the basic Euler method when applied to second order ODE problems. As is the case with the basic Euler method, users are advised to avoid using the symplectic Euler method.

See appendix C for details.

3.1.4.9.3 Beeman's Algorithm

Beeman's algorithm is a predictor-corrector algorithm tailored to second order ODEs. Its global error is second order in time, making it significantly better than the Euler techniques described above.

See appendix D for details.

3.1.4.9.4 Second Order Nyström-Lear (NL2)

The second-order Nyström-Lear techhnique is a Runge-Kutta-style technique tailored to second order ODEs. As the name suggests, it also exhibits a global error that is second order in time.

See appendix E for details.

3.1.4.9.5 Position Verlet

The position verlet method is a member of the verlet family of integrators, all of which exhibit a global error that is second order in time. The verlet algorithms alternate between advancing position and velocity at half step intervals. The position verlet algorithm advances velocity at the half step, which means that position verlet only needs one derivative call per integration step.

See appendix F for details.

3.1.4.9.6 Heun's Method

Heun's method can be viewed as either a second order Runge-Kutta integrator or as a simple predictor-corrector algorithm. It advances state as a full Euler step using the derivative at the start of the interval followed by a corrective Euler step using the average of initial and final derivatives.

See appendix G for details.

3.1.4.9.7 Midpoint Method

The midpoint method is another member of the second order Runge-Kutta family of integrators. It advances state as an Euler step to the midpoint of the interval followed by a full Euler step using the midpoint derivatives.

See appendix H for details.

3.1.4.9.8 Velocity Verlet

The velocity verlet method is a member of the verlet family of integrators. The velocity verlet algorithm advances velocity at step boundaries. Unlike the position verlet method, the velocity verlet needs two derivative calls per intergration step because of drag. Also unlike the position verlet method, the velocity verlet method is self-starting.

See appendix I for details.

3.1.4.9.9 Modified Midpoint (MM4)

This method is of unknown heritage. The name suggests that it exhibits an error that is fourth order in time. This is not the case. This method exists for backward compatibility only. Its use is not recommended.

See appendix J for details.

3.1.4.9.10 Fourth Order Adams-Bashforth-Moulton (ABM4)

The fourth order Adams-Bathforth-Moulton technique is a predictor-corrector algorithm for first order ODEs. This technique uses the four step Adams-Bashforth integrator as a predictor and the three step Adams-Moulton as a corrector. This technique exhibits a global error that is fourth order in time. It only requires two derivative calls per step, but does so at the expense of not being self-starting and at the expense of mishandling discontinuities in the derivatives.

See appendix K for details.

3.1.4.9.11 Fourth Order Runge-Kutta

The classical fourth order Runge-Kutta technique advances state by taking four steps per integration cycle. The first two steps advance state to the midpoint of the integration interval while the last two advance state to the end of the interval. As the name suggests, this technique exhibits a global error that is fourth order in time.

See appendix L for details.

3.1.4.9.12 Fourth Order Runge-Kutta-Gill

The fourth order Runge-Kutta-Gill method uses the same intermediate steps as does the classical fourth order Runge-Kutta technique but uses a different Butcher tableau.

See appendix M for details.

3.1.4.9.13 Fifth Order Runge-Kutta-Fehlberg

The Runge-Kutta-Fehlberg family of techniques were designed for use in adaptive integrators, integrators that change the step size to keep the error within some bound. These style integrators use a pair of Runge-Kutta integrators to provide different estimates of the integrated state. The fifth order Runge-Kutta-Fehlberg technique provided with the ER7 Utilities Integration model is not adaptive. It is the higher order member of the fourth and fifth order techniques used by an adaptive Runge-Kutta-Fehlberg 4/5 integrator.

See appendix N for details.

3.1.4.9.14 Eighth Order Runge-Kutta-Fehlberg

The eighth order Runge-Kutta-Fehlberg is the higher order member of the seventh and eighth order techniques used by an adaptive Runge-Kutta-Fehlberg 7/8 integrator. This technique, like the fifth order Runge-Kutta-Fehlberg provided with the ER7 Utilities Integration model, is not adaptive.

See appendix O for details.

3.1.4.10 Supported Initial Value Problems

The techniques listed above address one or more of the following types of initial value problems:

- First order ODE,
- Second order ODE,
- Second order generalized derivative ODE, and
- Second order Lie group ODE.

These are described in detail in the following paragraphs.

There is an obvious mismatch between the first and second order techniques in the above list. The reason is simple: Nobody has asked for integrators that solve the first order ODE equivalents of the second order generalized derivative and Lie group problems.

3.1.4.10.1 First order ODE

The first order ODE $\dot{x}(t) = f(x(t), t)$ describes the behavior of the problem to be solved. The state integrator for a first order ODE solver receives the vector $x(t)$ and its time derivative $v(t) = f(x(t), t)$ as arguments.

Most of the techniques listed in section 3.1.4.9 are first order ODE solvers. Others are specific to second order ODEs. While a first order ODE solver can be used to solve a second order ODE, the converse is not true. Those techniques that are specific to second order ODEs provide a surrogate when asked to provide a solver for a first order ODE.

3.1.4.10.2 Second order ODE

The second order ODE $\dot{x}(t) = v(t)$, $\dot{v}(t) = f(x(t), v(t), t)$ describes the behavior of the problem to be solved. The state integrator for a second order ODE solver receives the vectors $x(t)$, $v(t)$, and $a(t) = f(x(t), v(t), t)$ as arguments.

All of the techniques listed in section 3.1.4.9 can solve a second order ODE. Some of the listed techniques specifically address second order ODEs; the remaining techniques are first order ODE solvers that can easily be adapted to solving a second order ODE.

3.1.4.10.3 Second order generalized derivative ODE

The generalized second order ODE $\dot{x}(t) = g(x(t), v(t))$, $\dot{v}(t) = f(x(t), v(t), t)$ describes the behavior of the problem to be solved. As is the case with the basic second order ODE solver, the state integrator for a generalized derivative second order ODE solver receives the vectors $x(t)$, $v(t)$, and $a(t) = f(x(t), v(t), t)$ as arguments.

The function $g(x(t), v(t))$ maps generalized position and generalized velocity to the first time derivative of generalized position. Some integration techniques need an auxiliary function $h(x(t), v(t), a(t))$ that maps generalized position, generalized velocity, and the time derivative of generalized velocity to the second time derivative of generalized position. These two functions must be specified when the generalized derivative integrator is created.

All of the techniques listed in section 3.1.4.9 can solve a second order generalized derivative ODE.

3.1.4.10.4 Second order Lie group ODE

The generalized second order ODE $\dot{x}(t) = \alpha\, v(t) \bullet x(t)$, $\dot{v}(t) = f(x(t), v(t), t)$ describes the behavior of the problem to be solved. As is the case with the basic second order ODE solver, the state integrator for a generalized derivative second order ODE solver receives the vectors $x(t)$, $v(t)$, and $a(t) = f(x(t), v(t), t)$ as arguments.

In the above, the generalized position $x(t)$ is an element of a Lie group, the generalized velocity is an element of a Lie algebra for that Lie group, and α is a constant scale factor. The product $v(t) \bullet x(t)$ is performed in the context of the group action of the Lie group. This multiplication typically is not commutative in these Lie group problems: $a \bullet b \neq b \bullet a$ in general. Indeed, the Lie bracket $[a, b] = a \bullet b - b \bullet a$ typically is not zero, and typically plays a key role in the solutions to the problem.

The second order Lie group problems are obviously a special case of the second order generalized derivative problems described above. The rationale for these Lie group solvers is that they pay attention to the geometry of the problem. If a technique provides a Lie group solver it will inevitably perform much better than the corresponding generalized derivative solver.

This would be an academic exercise were it not for the fact that rotations in three dimensional space form a Lie group, the group $SO(3)$. The Lie group solvers were developed because of the relevance of this group to physics-based simulations and because the Lie group techniques offer marked improvements in accuracy over alternative formulations.

The Lie group solvers need two special functions that need to be specified at construction time. The *expmap* function approximates the Lie group's exponential map, while the *dexpinv* function approximates the Lie group's pullback function.

Only some of the techniques listed in section 3.1.4.9 can solve a second order Lie group ODE. The second order generalized derivative solver should be used in lieu of the Lie group solver in the case of a technique that does not provide a Lie group solver.

3.1.4.11 Integrator Constructor Factory

The ER7 Utilities Integration model provides an integrator constructor factory that simplifies the specification of an integration technique. The factory constructs an integrator constructor that corresponds to one of the supported techniques. All the user needs to specify is an element of the enumeration of supported integration techniques.

3.1.4.12 Support for Generalized Second Order Integrators

To illustrate how the functions needed by the generalized derivative and Lie group integrator are to behave, the ER7 Utilities Integration module provides implementations of the four functions needed by a second order ODE left quaternion representation of orientation. It is intentional that this is exactly the functionality needed by JEOD.

3.1.5 JEOD Integration Model

The JEOD Integration Model builds on the functionality provided by the ER7 Utilities Integration model.

3.1.5.1 JEOD Integration Group

A JEOD integration group extends the concept of an ER7 Utilities integration group. The ER7 Utilities concept is rather basic. It has to be so as to accommodate both the Trick and JEOD concepts of integration. This extension is limited to core concepts. It does not cover dynamics, as this is the purview of the JEOD dynamics models. The *Dynamics Manager Model* [5] in turn extends the JEOD integration group to cover dynamics.

3.1.5.2 Generalized Second Order ODE

As mentioned above, a second order Lie group problem is a special case of a second order generalized derivative problem. Some integration techniques only provide a second order generalized derivative integrator. The default in JEOD 3.0 is to use the second order Lie group integrator if available. If it's not available, the second order generalized derivative solver can always be used as a backup. The Integration Model provides mechanisms for dealing with a mismatch between requested and available techniques.

3.1.5.3 Restartable State Integrator

C++ does not provide garbage collection, making properly managing resources a key problem for C++ programs. One approach is to put the burden entirely on the programmer: make the programmer responsible for allocating and releasing all needed resources such as allocated memory. A better approach is to use techniques such as the non-intuitively named RAII ("Resource Allocation Is Initialization") design pattern to remove this burden from the programmer. This design pattern incorporates resource allocation and deallocation into the object design.

Because state integrators are allocated by an integrator constructor, this makes the state integrators used in JEOD subject to the very problem that the RAII design pattern solves. The Integration Model provides class templates that solve the allocation problem in a generic sense and provides three instances of these class templates that specifically address the three kinds of initial value problems found within JEOD.

3.1.5.4 Time

The ER7 Utilities Integration model provides a very abstract concept of time. The JEOD Integration Model extends this abstract concept, but only to the point needed by the model itself. The *Time Model* [14] further extends this JEOD integration time to form the JEOD time manager.

Users can make the JEOD time manager change rate. This can wreak havoc with predictor-corrector methods unless these integrators are reset. The Integration Model implements a concept of a time change subscriber so that interested parties can respond to changes in JEOD time. The JEOD integration group is a time change subscriber.

3.1.5.5 JEOD-Specific Integration Techniques

All of the integration techniques described in section 3.1.4.9 can be used in JEOD-based simulations. JEOD provides two additional techniques aimed at long arc integration problems. These two techniques are described below.

3.1.5.5.1 Gauss-Jackson

The Gauss-Jackson integrator provided with JEOD 3.0 is a non-adaptive (fixed step size) predictor-corrector solver for second order ODEs. Users have control over the order of the technique and tolerance targets of the corrector algorithm. This technique performs particularly well compared to other techniques when applied to nearly circular orbits.

See appendix P for details.

3.1.5.5.2 LSODE

The Livermore Solver for Ordinary Differential Equations (LSODE) uses a number of integration techniques to solve intial value problems for first order ODEs. Some of the solvers used by LSODE are adaptive. Users have control over the solver algorithm, the order technique, and the tolerance targets. This technique performs particularly well compared to other techniques when applied to markedly non-circular trajectories.

See appendix Q for details.

3.2 Key Algorithms

This section summarizes key algorithms used in the implementation of the Integration Model.

3.2.1 Integration Techniques

Each integration technique is described in detail in an appendix to this document. This section summarizes key characteristics of the techniques. Table 3.1 presents a summary of the integration techniques.

Table 3.1: Integration Technique Characteristics

Technique	Accuracy (order)	Needs Priming	Number Stages Per Cycle [a]	First Step Derivatives [b]	er7_utils Enum Value	Trick Enum Value
Euler	$O(\Delta t)$	No	1	Yes	`Euler`	`Euler`
Symplectic Euler	$O(\Delta t)$	No	1	Yes	`SymplecticEuler`	`Euler_Cromer`
Beeman's Algorithm	$O(\Delta t^2)$	Yes	2	Yes	`Beeman`	N/A
Second order Nyström-Lear	$O(\Delta t^2)$	Yes	2	No	`NystromLear2`	`Nystrom_Lear_2`
Position verlet	$O(\Delta t^2)$	No	2	No	`PositionVerlet`	N/A
Heun's method	$O(\Delta t^2)$	No	2	Yes	`RK2Heun`	`Runge_Kutta_2`
Midpoint method	$O(\Delta t^2)$	No	2	Yes	`RK2Midpoint`	N/A
Velocity verlet	$O(\Delta t^2)$	No	2	Yes	`VelocityVerlet`	N/A
Modified midpoint method	$O(\Delta t^3)$	No	2	Yes	`ModifiedMidpoint4`	`Modified_Midpoint_4`
4th order Adams-Bashforth-Moulton	$O(\Delta t^4)$	Yes	2	Yes	`AdamsBashforthMoulton4`	`ABM_Method`
Runge-Kutta 4	$O(\Delta t^4)$	No	4	Yes	`RungeKutta4`	`Runge_Kutta_4`
Runge-Kutta-Gill 4	$O(\Delta t^4)$	No	4	Yes	`RKGill4`	`Runge_Kutta_Gill_4`
Runge-Kutta-Fehlberg 4/5	$O(\Delta t^5)$	No	6	Yes	`RKFehlberg45`	`Runge_Kutta_Fehlberg_45`
Runge-Kutta-Fehlberg 7/8	$O(\Delta t^8)$	No	12	Yes	`RKFehlberg78`	`Runge_Kutta_Fehlberg_78`
Gauss-Jackson	var.	Yes	var.	Yes	N/A	N/A
LSODE	var.	var.	var.	Yes	N/A	N/A

[a]This column indicates the number of steps per cycle after the technique is primed. Techhnique that needs priming typically exhibit different behavior during priming.

[b]This column indicates whether derivative data must be provided to the integrator on the first step of the integration cycle. Some integration techniques do not use the derivatives on the first step of an integration cycle. As computing derivatives is typically a rather expensive computational process, the computation of those derivatives can be bypassed if they aren't used.

3.2.2 Generalized Derivative Second Order ODEs

3.2.3 Lie Group Integration

3.3 Detailed Design

The classes and methods of the Integration Model are described in detail in the *Integration Model API* [4].

The remainder of this section is TBS (to be supplied).

3.4 Interactions

3.4.1 JEOD Models Used by the Integration Model

The Integration Model uses JEOD models to manage memory and to update time.

Several model classes allocate memory and deallocate memory using the *Memory Management Model* [6]. The IntegratorConstructorFactory creates an instance of a class that derives from the IntegratorConstructor class. Each integrator constructor creates instances of state and time integrators. State integrators allocate memory for internal storage. The internal storage allocated by the state integrators is freed when the state integrators are destroyed. The freeing of the instances of integrator constructors, state integrators, and time integrators is the responsibility of the objects that created these instances rather than the Integration Model.

The time integrators update the *Time Model* [14] by invoking the `TimeManager::update` method with the interpolated simulation time at the end of each intermediate step.

3.4.2 Use of the Integration Model in JEOD

Two JEOD models use the Integration Model to integrate vehicle states over time:

- The *Dynamics Manager Model* [5] uses one of the Integration Model integrator constructors to create a time propagator and to create state integrators for each dynamic body in the simulation. The time integrator is owned by the dynamics manager while the state integrators are owned by the dynamic bodies. In the event that the dynamics manager is lacking a provided integrator constructor, the dynamics manager uses the Integration Model's integrator constructor factory to create an integrator constructor based on the integration option in a Trick integration structure.

  Once the simulation is running, the dynamics manager propagates time using the time integrator and directs each root dynamic body to propagate its state. As the time and state integrators were created by a single integrator constructor, this approach ensures consistent state integration and time propagation across a simulation.

- The *Dynamic Body Model* [7] uses the integrator constructor supplied by the dynamics manager to create rotational and/or translational state propagators. While the simulation is running, a dynamic body uses its constructed state propagators to propagate the body's rotational and translational states.

3.4.3 Interactions with Trick

The Integration Model does not interact directly with Trick. In JEOD, that is the job of the Dynamics Manager Model. The Integration Model does provide interfaces that support these Trick/JEOD integration interactions.

Trick uses INTEGRATOR structures and outputs from integration class jobs to drive the integration process. The `first_step_deriv` flag in the INTEGRATOR structure determines whether the derivative functions are to be called on the first step of each integration cycle. Each IntegratorConstructor defines a `first_step_second_derivs_flag` method to support the setting of this flag. Trick deems the integration cycle to be complete for a given integrator when the integrator returns a step number of zero. The Integration Controls supports this interface by having the `integrate` methods return the step number. The step number in integration controls is advanced only when the integration group (and thereby all of the two-state integrators contained therein) and the time integrator indicate that they have met their targets.

3.4.4 Interaction Requirements on the Dynamics Manager Model

The simulation's dynamics manager propagates the contents of all integration groups (time and all dynamic bodies within that group) registered with the dynamics manager. Time must be propagated within the integration cycle as some derivative functions such as non-spherical gravity depend on time. The Integration Model explicitly provides the ability to meet this need. The Dynamics Manager must use this provided capability to have the integrated vehicle states best reflect reality.

3.4.5 Interaction Requirements on the Dynamic Body Model

The dynamic bodies in a simulation use the Integration Model to propagate states over time. The Integration Model provides a generic capability to integrate generalized position and velocity. The user of a state integrator must properly tailor its use of the integrator to the problem at hand. This tailoring includes initialization and operation. The integrator must be constructed properly at initialization time and must be fed with properly constructed state and derivative vectors during operations.

3.5 Inventory

All Integration Model files are located in `${JEOD_HOME}/models/utils/integration`. Relative to this directory,

- Model header and source files are located in model `include` and `src` subdirectories. See table ?? for a list of these configuration-managed files.

- Model documentation files are located in the model `docs` subdirectory. See table **??** for a list of the configuration-managed files in this directory.

Chapter 4

User Guide

The User Guide is divided into three instruction sets:

1. **Instructions for Simulation Users**. This instruction-set contains a description of how to modify Integration Model variables within an existing simulation, including a discussion of the input file and an overview of how to interpret (but not edit) the S_define file.

2. **Instructions for Simulation Developers**. This instruction-set builds on the previous set, adding information on the necessary configuration of the Integration Model within an S_define file, and the creation of standard run directories.

3. **Instructions for Model Developers**. This instruction-set builds on the previous set, and adds information on the potential for extending the model to perform tasks that are beyond its current abilities, such as adding new integration algorithms.

4.1 Instructions for Simulation Users

A simulation user in general does not concern themselves with the Integration Model. The integrators are often created during simulation initialization, which means that the internal data are not accessible to the Trick logging mechanisms. The model works behind the scenes, propagating the states of the simulated vehicles based on the forces and torques acting on the vehicles.

The simulation's dynamics manager (see *Dynamics Manager Model* [5]) directs the integration of all dynamic bodies registered with it. Each dynamic body has its state derivatives computed, based on the forces and torques acting on it. The Integration Model propagates states according to the state derivatives sent to it. Properly formulating these state derivatives is one of the jobs of the *Dynamic Body Model* [7].

Two of the factors affecting the accuracy of the propagated state are the choice of the simulation time step, and the algorithm used to numerically propagate the state. Every integration algorithm has some optimal time step for a given problem. Running slower than this optimal step size reduces accuracy because of limitations inherent in the algorithm. Running faster than this optimal step size degrades accuracy because the of limitations inherent in the use of IEEE floating point arithmetic.

For spacecraft simulation that include a model of the flight software, the operation of the flight software often places an upper bound on the simulation time step. The flight software time step is often shorter than the optimal time-step (viewed from the perspective of accurately propagating state), resulting in a suboptimal choice for integration rates. Choosing to run the simulation at a frequency even greater than the flight software frequency will further degrade accuracy and performance, while it is not possible to run more slowly than flight software requires. In short, changing the simulation time step is often not an available option.

There is one marked exception to the above discussion. Ofttimes the behavior of a failed vehicle must be analyzed. As the flight software is not operating in these scenarios, the flight software time step is not a constraint. These scenarios can be run at a frequency slower than the flight software frequency. Doing so will decrease CPU utilization and may well increase accuracy.

The appropriate choice of the integration algorithm can enhance accuracy and performance. The burden of selecting the appropriate compromise between accuracy and performance falls on the simulation analyst and integrator. To aid in making that decision, a synopsis of the techniques available in JEOD v5.1 is presented in table 4.1. The second and third columns specify the number of calls to the integration and derivative class jobs per simulation time step. The number of times these functions are called is one of the key factors that determine the amount of CPU time consumed by a simulation. The last four columns in the table present the three-sigma error bounds on a circular orbit simulation integrated with $\omega\Delta t = 0.0562°$. For a 400 km LEO orbit, that corresponds to integrating at 1.15 Hz. These error bounds are depicted graphically in Figure 4.1.

Table 4.1: Integration Technique Performance Characteristics

Technique	**History length**	**Stages per cycle**	**Accuracy (order)**	**Error, 1 orbit**	**Error, 3 orbits**	**Error, 10 orbits**	**Error, 30 orbits**	**Error, 100 orbits**
Euler	0	1	$O(\Delta t)$	400 km	3000	10000	10000	20000 km
Symplectic Euler	0	1	$O(\Delta t)$	10 km	10	10	10	10 km
RK2	0	2	$O(\Delta t^2)$	0.05 km	0.2	0.5	2	6 km
RK4	0	4	$O(\Delta t^4)$	0.002 mm	0.007	0.04	0.2	0.9 mm
ABM4	4	2	$O(\Delta t^4)$	0.002 mm	0.006	0.03	0.2	2 mm
Beeman's Algorithm	1	2	$O(\Delta t^2)$	6 m	20	60	200	600 m

Note that the Euler techniques are quite inaccurate, and their use is dicouraged. In particular, use of the basic Euler technique is very strongly discouraged. It is included with JEOD only because understanding how the basic Euler technique works is critical for understanding how the more advanced techniques work. The symplectic Euler technique may be useful in situations in which the integration must be performed at such a high frequency that even a two-stage integrator would be prohibitively expensive.

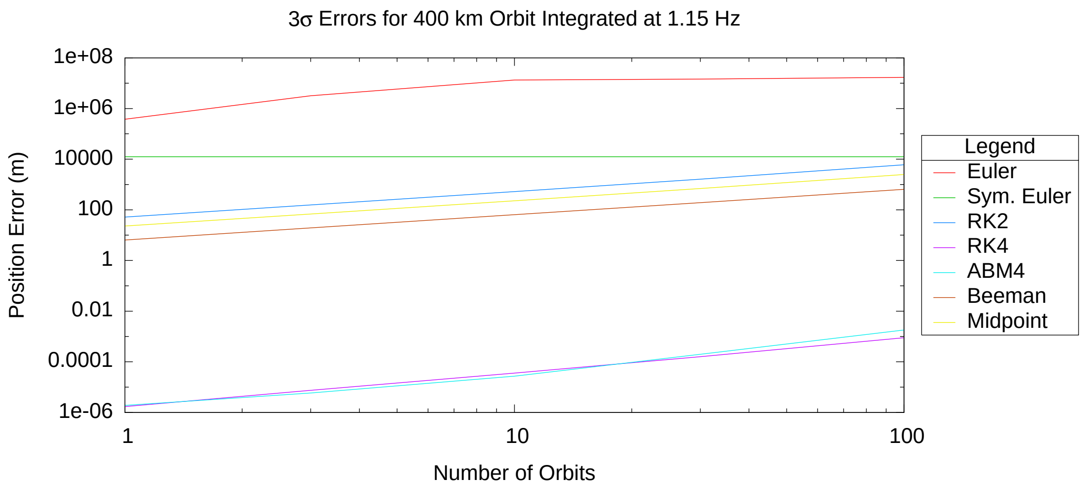


Figure 4.1: Integration Technique Performance Characteristics

4.1.1 Changing the Integrator

The assignment of which integration algorithm will be used is almost always made at the input file level. Look for one of the following specifications:

```
dynamics.manager_init.integ_constructor = trick.<integration-method>()
dynamics.manager_init.integ_constructor = <sim-object>.<integration-method>
dynamics.manager_init.jeod_integ_opt = trick.Integration.<integration-method>
dynamics.manager_init.sim_integ_opt = trick.sim_services.<integration-method>
```

These are listed in order of preferred practice.

1. **integ_constructor method with local instantiation**. If this method has been used, the user will have to look in the S_define to identify which integrator constructors have been included therein. Look for lines such as

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

   The integrator can be changed to any of the other integrators for which the appropriate header file has been included. Specifications for `<integration-method>` are:

   - ABMIntegratorConstructor
   - BeemanIntegratorConstructor
   - EulerIntegratorConstructor
   - GaussJacksonIntegratorConstructor
   - RK2IntegratorConstructor
   - RK4IntegratorConstructor
   - SymplecticEulerIntegratorConstructor

Remember to add the () after the specification; this is actually a method call to create a new instance of that type of integrator.

Also be sure to check in the S_define file to ensure that an instance of the IntegratorConstructor has not already been made. If it has, use option #2 (below) instead.

If the desired integrator is not available, the user may be able to use option #3 (below).

2. **integ_constructor method using S_define instantiation**. If this method has been used, the user will have to look in the S_define to identify which integrator constructors have been declared therein (note - these will likely be in the simulation object `<sim-object>`). Any declared constructor can be substituted by replacing `<integration-method>` with the name of the instance of another integrator constructor.

   Note that in order for the integrator constructor to be declared, its header file must previously be `##include` in the S_define, so option #1 will, in principle, also work. However, if one instance of the integrator constructor already exists, another should not be made, so option #2 is preferred **if** the simulation developer (i.e. the author of the S_define file) has already made such an instance.

   If the desired integrator is not available, it may be possible to change to option #3 (below).

3. **jeod_integ_opt method**. This method can be used with the following options for `<integration-method>`:

   - Euler
   - Symplectic Euler
   - Beeman
   - RK2
   - RK4
   - ABM4

4. **sim_integ_opt method**. As a last resort, this method can be used with the following options for `<integration-method>`:

   - Euler
   - Euler_Cromer
   - Runge_Kutta_2
   - Runge_Kutta_4
   - ABM_Method

4.1.2 Changing the Integration Rate

The integration rate is typically hard-coded within the simulation. Look towards the end of the S_define file for an entry such as

```
IntegLoop sim_integ_loop (DYNAMICS) <simulation object list>;
```

or

```
integrate (DYNAMICS) <simulation object list>;
```

In either case, the value in parentheses (sometimes a number, sometimes a parameter) is the integration period. if a parameter, it will be defined somewhere in the S_define, most likely near the top.

```
#define DYNAMICS        1.00  // Dynamics interval
```

There are now two choices for changing the integration rate:

1. Change this value; this requires editing the S_define and recompiling.
2. Because this value is the simulation-time, and the actual dynamics depend on the dynamic-time, it is possible to change (in the input file) the actual dynamic-time-step without altering the simulation-time-step by altering the rate at which the dynamic-time advances with simulation-time.

   For example,

   ```
   jeod_time.time_manager.dyn_time.scale_factor = 0.5
   ```

   will make the dynamic-time proceed only half as fast as simulation-time, so a simulation-time rate of 1.0 seconds will equate to a dynamic-time rate of 0.5 seconds.

   Users attempting this change should be familiar with the distinction between these two concepts of time. See the *Time Model* [14] for more details.

4.1.3 Moving the Integrated Object From One Group to Another

In rare situation, the simulation may be integrating two different groups of objects at different rates and.or using different intergators. It is possible to switch an object from one group to another, using the *add_sim_object* method.

To identify whether multiple groups are being used, look to the end of the S_define for multiple instances of code such as:

```
JeodIntegLoopSimObject <loop_name> (
   <rate>,
   <constructor>,
   <group>,
   <time-manager>,
   <dyn_manager>,
   <gravity_model>,
   <models to be integrated>,
   NULL);
```

If this is found, then objects may be moved from one group to another:

```
<loop_name>.integ_loop.add_sim_object(<name of object>)
```

Note - `<loop_name>` is the name of the group that the object is moving *to*.

4.1.4 Editing the Specifications of the Integrators

Of the integrators that JEOD provides, only the Gauss-Jackson method has user-definable specifications.

4.1.4.1 Gauss Jackson Parameters

For Gauss-Jackson, the following values can be set:

- The order of the integrator. This defaults to 8, and can be set from 1 to 16.
- The maximum time-step to use on the primer. This defaults to -1.0; indicating that the primer time-step should be the same as the cycle time-step and tour time-step. If the tour time-step is too large for the RK4 integrator (RK4 serves as the priming integrator, it is used for the first few steps) to handle should this value be set. Set it to a value that is appropriate for the limitations of RK4.
- Whether or not to perform the convergence test after the correction phase. This defaults to false.
- The convergence criterion. This defaults to 1.0×10^{-9}. This value is only used if the convergence test is performed.
- The maximum number of convergence iterations. This defaults to 10. It represents the maximum number of times that the state-correction phase can be implemented in one integration cycle. For more precise work, set the convergence criterion smaller, and the maximum number of iterations larger.

Example:

```
gj_integrator = trick.GaussJacksonIntegratorConstructor()
gj_integrator.order = 10
gj_integrator.max_rk4_step= 1.0
gj_integrator.perform_convergence_test = true
gj_integrator.convergence_criterion = 1E-10
gj_integrator.max_corrections_iterations = 15
```

Note - these values are used in the construction of the integrator. Once the constructor has been created, changes to these values will have no effect. Changes can only be made at the start of the simulation.

The verification simulation *SIM_GJ_test* (on page 47) provides a quantitative illustration of the effect of changing these parameters for a very simple scenario.

4.2 Instructions for Simulation Developers

Instructions are provided for simulation developers working with the Trick13 simulation engine. JEOD v5.1 is not compatible with versions of Trick older than 13.0.

4.2.1 Populating Integration Groups

The purpose of integration groups is to allow the simultaneous integration of models of disparate fidelity requirements within a single simulation. In most applications, this tool is not needed, in which case the default behavior – in which there is only one group – is assumed. Each group must define its own integrator following the steps in Section 4.2.2.

4.2.1.1 Single Group Default Settings

Without the need for multiple groups, the integration command is straightforward:

```
IntegLoop sim_integ_loop (DYNAMICS) <simulation object list>;
```

This command comes at the end of the S_define. The argument (e.g. DYNAMICS) is the integration rate (note - this is based on the simulation time, not the dynamic time; for the difference, see the Time Model documentation [14]). Following the argument is a comma-separated list of all of the simulation objects that need integrating.

An equivalent command is:

```
integrate (DYNAMICS) <simulation object list>;
```

4.2.1.2 Multi-Group Settings

The prefered method for setting multiple groups is to use the standard module, *JEOD_S_modules/integ_loop.sm*. In the S_define file, include this file:

```
#include "JEOD_S_modules/integ_loop.sm"
```

Unlike most of the other sim-modules in the JEOD_S_modules directory, this one does not create a simulation object, it only defines its contents.

Declare an instance of the DynamicsIntegrationGroup somewhere in the S_define, for example in a simulation object called *integ_object*. Note that this instance is not going to be used directly, so one instance will suffice for as many groups as are desired.

```
##include "dynamics/dyn_manager/include/dynamics_integration_group.hh"
...
DynamicsIntegrationGroup group;
```

At the end of the S_define, create one instance of the loop object defined in the integ_loop file for each intended group.

The structure of this argument is as follows:

```
JeodIntegLoopSimObject <name of loop-object> (arguments)
```

The arguments comprise the following list:

1. The integration-rate. This is often pre-defined, in which case only the name is needed.
2. The instance of the desired IntegratorConstructor. For example, to use an Euler integration, declare an instance of EulerIntegratorConstructor in integ_object:

   ```
   EulerIntegratorConstructor euler_constr;
   ```

   and add the argument *integ_object.euler_const*.
3. An instance of the IntegrationGroup class (e.g. *integ_object.group*).
4. The TimeManager object for the simulation. There should be only one, if using the standard sim-modules, it will be *jeod_time.time_manager*.
5. The Dynamics Manager (DynManager) object. Again, there should be only one in the simulation; if using the standard sim-modules, it will be *dynamics.dyn_manager*.
6. The gravity manager (GravityManager) object. Again, there should be only one in the simulation; if using the standard sim-modules, it will be *env.gravity_model*.
7. The address of the first simulation object to be integrated in the group. For example, with a simulation-object called vehicle_1, *&vehicle_1*.
8. Repeat the previous step to add the addresses of as many simulation objects as go with this group.
9. Finish the argument with NULL to indicate that all objects have been entered.

Note that an empty group may be created by specifying no simulation objects.

These examples are taken from *environment/ephemerides/verif/SIM_prop_planet_T10*.

```
JeodIntegLoopSimObject fast_integ_loop (
   FAST_DYNAMICS,
   integ_constructor.selected_constr, integ_constructor.group,
   jeod_time.time_manager, dynamics.dyn_manager, env.gravity_manager,
```

```
    &sun, &jupiter, &saturn, NULL);


JeodIntegLoopSimObject slow_integ_loop (
    SLOW_DYNAMICS,
    integ_constructor.selected_constr, integ_constructor.group,
    jeod_time.time_manager, dynamics.dyn_manager, env.gravity_manager,
    NULL);
```

Once created, objects may be moved from one group to another with the command

```
<loop_name>.integ_loop.add_sim_object(<name of object>)
```

For example, from one of the SIM_prop_planet input files
(*environment/ephemerides/verif/SIM_prop_planet_T10/SET_test/RUN_switch_integ/input.py*),

```
  slow_integ_loop.integ_loop.add_sim_object(saturn);
```

It is not necessary to remove an object from its previous integration group. An object can only belong to one group so is removed automatically.

4.2.1.3 Caution on Groups with Different Rates

While having integration groups with different rates is a valuable tool, it can lead to some unfortunate consequences if misused. The user should be aware that making a direct comparison of logged values from these two groups may result in inconsistencies. Each group will be integrated through its complete tour before the next group starts. The objects in the group integrated at lower frequency will sit with the same state while the higher frequency group catches up. Looking at relative states between such entities may lead to inappropriate conclusions. When the higher rate is not an integer multiple of the lower rate (e.g. one at 5Hz and one at 7Hz), the times that the two groups are providing time-consistent data can be few and far-between.

This is particularly important when switching an object from one group to another. When the object switches, its current state is maintained even if it is invalid at the current time. For example, suppose some vehicle, was coasting with an integration period of 10 seconds. At t=126s, the vehicle is activated, and now needs integrating at the flight software rate, say 50Hz. Transitioning immediately to the higher rate group would put the vehicle at t=126s with the state it had at t=120s (its last integration). From here (t=126s), that wrong state will be propagated.

4.2.2 Setting the Integration Method

In order to use an integrator, it must be available to the simulation engine at compilation time. There are two ways by which this can be done:

1. inclusion of the integrator from a standard suite, via an enumerated list, or

2. deliberate declaration of the integrator-constructor in the S_define.

Having the integrator available without deliberate inclusion may be easier to implement, but this process has the significant drawback that the code for that integrator must then be compiled as a part of the simulation whether it is desired or not. This time-consuming model is suboptimal, and while it is supported in the current release, it may be deprecated later. The recommended practice, therefore, is to instantiate the integrator-constructors directly, in either the S_define or the input file.

To provide backward compatibility, the enumerated list of integrators is still provided, as is the conversion from the Trick enumerated list. However, there are no plans to add new integrators to the list. Consequently, the only way to access the Gauss-Jackson method is through the recommended practice of direct declaration.

Furthermore, when using multiple integration groups, the process for defining which integrator to use with which group requires that the integrator-constructor be available for passing as an argument. In this case, the recommended method is required.

The pre-loaded integrators are available via an enumerated list. The enumeration value can be accessed from either the JEOD list or specified via the trick simulation interface method using the comparable Trick list.

There are three methods available for selecting an integrator, as discussed below.

4.2.2.1 Recommended Practice

Setting the Dynamics Manager *integ_constructor* value is the priority method for defining the integrator. It is this value that is tested first.

There are two ways to do this:

1. **The easy way:**

   (a) Include the header file in the S_define, e.g.

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

   (b) then instantiate the Integrator Constructor by assignment in the input file, e.g.

   ```
   dynamics.manager_init.integ_constructor = trick.RK4IntegratorConstructor()
   ```

2. **The more traditional way:**

   (a) Instantiate an instance of the integrator-constructor in the S_define.

   (b) Point the DynManagerInit value integ_constructor to the instance of the integrator-constructor

Example:

In the S_define, create a sim-object called *integ_sim_object*, that includes an instance of a RK4IntegratorConstructor called *rk4*.

```
##include "utils/integration/include/rk4_integrator_constructor.hh"
class IntegSimObject: public Trick::SimObject {
...
RK4IntegratorConstructor rk4;
...
}
IntegSimObject integ_sim_object;
```

In the input file, declare the *integ_constructor* to be this new constructor (note - although the *integ_constructor* is a pointer, Python can assign it without having to take the address)

```
dynamics.manager_init.integ_constructor = integ_sim_object.rk4
```

4.2.2.2 Allowable non-optimal practice

If the integ_constructor pointer is NULL (i.e. has not been set), then the next-preferred method is to use the *jeod_integ_opt* specification. The allowable targets for this are the enumerated items in Table 4.2, each prefaced with `trick.Integration.` .

Example:

```
dynamics.manager_init.jeod_integ_opt = trick.Integration.RK4
```

Table 4.2: Integration Technique Enumeration

Integrator	Enumeration	Trick Equivalent
Euler	1	Euler
Symplectic Euler	2	Euler_Cromer
Beeman	3	N/A
RK2	4	Runge_Kutta_2
RK4	5	Runge_Kutta_4
ABM4	6	ABM_Method
Gauss-Jackson	N/A	N/A

4.2.2.3 Deprecated Practice

It is still possible to declare the method using the Trick enumerated list, and that practice is still widely used, although it is no longer supprted. This method is only followed when both the *integ_constructor* and *jeod_integ_opt* are not set. It is included here to provide simulation developers with information on how to interpret this command if seen in other simulations.

The *sim_integ_opt* value in the dynamics manager initializer can be set to an element in the Trick integration list. Note that only those elements in the Trick integration list that appear in Table 4.2 are allowable entries for JEOD integration. Each entry must be prefaced with `trick.sim_services`.

Example:

```
dynamics.manager_init.sim_integ_opt = trick.sim_services.Runge_Kutta_4
```

The four examples above will all produce the same result.

4.3 Instructions for Model Developers

This section has two primary components: how to add new integration techniques to the Integration Model, and how to use the Integration Model in other extensions of JEOD(see *Using This Model in JEOD Extensions* (on page 35)).

4.3.1 Adding New Integration Techniques

A wide range of numerical integration techniques exist, only a few of which are implemented by JEOD. This section describes what needs to be done to implement a numerical integration algorithm using the Integration Model architecture.

A model extender will typically need to write four classes:

- A class that derives from the TwoStateIntegrator class. This derived class will implement the desired numerical integration algorithm.

  The new TwoStateIntegrator class needs to define

  - A non-default constructor for this class, including arguments typically specifying
    - The number of elements in the zeroth-derivative (e.g. position) specification, and its derivative,
    - The number of elements in the first-derivative (e.g. velocity) specification, and its derivative,
    - An instance of a method used to compute the derivative of the zeroth-derivative state (typically from the zeroth- and first-derivative states). This method originates from the template structure in the *RestartableStateIntegrator* class, and is discussed further in the next section (see *Using This Model in JEOD Extensions* (on page 35)),
    - A reference to the Integration Controls appropriate to the integrator.

    This constructor should then, in turn, call the base-class TwoStateIntegrator constructor with the same set of arguments.
  - An `integrate` method that implements the algorithm. This method must have the following arguments:
    - A double representing the time-step (this is the cycle-time-step, for the dynamic-time)

    * An unsigned integer representing the cycle-stage to which the integrator is moving, the target-stage. This should come into the integrator from the controls, via the integration group. That way, all integrators in the group are assured to have the same target.
    * Four double-pointers representing, in order, the time-derivative of the first-derivative-state, the first-derivative-state, the time-derivative of the zeroth-derivative-state, and the zeroth-derivative-state. These should all point to arrays appropriately sized according to the values input into the constructor (first two arguments, outlined above).
  - Data members to contain internal storage needed by the algorithm,
  - A constructor and destructor to allocate and release memory,
  - A `delete_self` method to make an instance delete itself,
  - A `reset` method to make an instance of the class reset itself. This method can be empty if the integration algorithm is self-starting, but must be defined.

- A class that derives from the TimeIntegrator class. This derived class will advance time in a manner consistent with the numerical integration algorithm.

  The new TimeIntegrator class needs to define

  - An `integrate` method that advances time per the dictates of the desired numerical integration algorithm. The time integrator and state integrator must march in unison.
  - A `delete_self` method to make an instance delete itself,
  - A `reset` method to make an instance of the class reset itself.
  - a non-default constructor that takes a reference argument to the appropriate Integration Controls, and passes that on through to the TimeIntegrator non-default constructor.

- A class that derives from the IntegrationControls class.

  There are no required methods for the new class (the base class, IntegrationControls, is an instantiable class); however, the following methods may be necessary:

  - If the new integrator represents a multi-cycle process, the new class must set the `multi_cycle` flag to true in the constructor, and define
    * A `start_integration_tour` method if any additional processes are required when a new integration tour (not just an integration cycle) starts.
    * An `end_integration_cycle` method that tests for completion of the integration tour at the end of each integration cycle, and provides a suitable location for processes that occur only at the end of an integration cycle or an integration tour. The return value from this method should be *true* if the tour has completed, and *false* otherwise.
  - A `reset` method if any additional verification or assignments need to be made when the integrator is reset (default method is empty).
  - While the `integrate` method is a virtual method (so can easily be redefined for the new class), note that this method controls the entire sequencing of the integration process and is critically important. It has been carefully constructed to provide the flexibility to accommodate a very broad array of integration techniques; it may be easier to tweak the integration algorithm to fit into this existing framework than to rewrite the framework.

Note -

Integrators flagged as multi-cycle get two additional method calls (`start_integration_tour` and `end_integration_cycle`) regardless of how many cycles they take per tour. This feature can be useful for more than just monitoring multiple cycles. For example, the Beeman and ABM4 methods are multi-step, single-cycle methods, but the multi-cycle flag is set to assist with the priming process. In the class PrimingIntegrationControls, the `start_integration_tour` method is used to monitor the priming process of these integrators; once priming is complete, the multi-cycle flag is reset to false, and the method is not revisited.

- A class that derives from the IntegratorConstructor class. This derived class will create instances of the above three classes. The following methods must be defined:
  - A `create_two_state_integrator` method. This method takes the four arguments listed above for the constructor of the TwoStateIntegrator class. These values are passed directly through to the instantiation of the appropriate TwoStateIntegrator-derived class. Note - these values are set externally to the Integration Model with default values within JEOD; for more information, see *Using This Model in JEOD Extensions* (on the current page)).
  - A `create_time_integrator` method. This method takes the integration controls reference as an argument, and passes it through to the instantiation of the appropriate TimeIntegrator-derived class.
  - A `create_integration_controls` method. This method takes no arguments, and instantiates the appropriate IntegrationControls-derived class.
  - A `delete_self` method. This method deletes the Integrator Constructor itself.

General note -

While the constructor must always create instances of a two-state integrator, a time integrator, and an integration controls, existing classes can be re-used if the functionality is identical. Examples:

- Most integrators use the default IntegrationControls.
- The Symplectic Euler method uses the time integrator from the Euler method, since their procession in time are equivalent.

The ability to extend the model was tested in test Integration_4.

4.3.2 Using This Model in JEOD Extensions

Within JEOD, the Integration Model is purely used for the integration of a six-degree-of-freedom body, but the actual mathematical models within the Integration Model are far more generic, and can be applied to any second-order system. Of particular note is the recognition that the first-derivative-state, while intrinsically related to the zeroth-derivative-state, does not have to be a time-derivative of it. The example already in JEOD is the integration of the angular state, where the orientation and rate of orientation change are specified as quaternions, while the angular rate and rate of change of angular rate are specified as Cartesian vectors.

This same system is therefore useful in many other applications, such as a Hamiltonian system utilizing generalized momenta and generalized coordinates as the two states. Indeed, the two states may be completely unassociated as time-derivatives.

In JEOD, the Simple6DofDynBody (see the Dynamics Body documentation [7]) implements two integration generators, one for the translational state, and one for the rotational state. These are instantiated in *dynamics/dyn_body/simple_6dof_dyn_body.hh*:

- `SimpleRestartableIntegrator<3> trans_integ_generator;`
- `RestartableIntegrator<4, 3, Quaternion::compute_left_quat_deriv> rot_integ_generator;`

These structures are both templatized in *restartable_state_integrator.hh*, with the SimpleRestartableIntegrator a restricted case of the more general RestartableIntegrator.

In the latter, general, implementation, the arguments are:

1. The number of entries in the first state array (state[0], referred to as the zeroth-derivative-state in JEOD applications)
2. The number of entries in the second state array (state[1], referred to as the first-derivative-state in JEOD applications)
3. The specified method for creating the derivative of state[0] from state[0] and state[1].

With the integrator generators so specified, then control passes to the IntegrableObject (in JEOD, this is usually a Simple6DofDynBody, which is a DynBody, which is an IntegrableObject (among other things)). Somewhere in the creation of the IntegrableObject-derived instance, (for a Simple6DofDynBody, this is in the method *Simple6DofDynBody::create_integrators*), the generator method *create_integrator* is called, passing arguments of the appropriate instance of the IntegratorConstructor and IntegrationControls that will be used to integrate the state of this object.

The generator (an instance of the RestartableIntegrator class) method *create_integrator* calls the IntegratorConstructor method *create_two_state_integrator*, with 4 arguments: the three arguments used in the template discussed above, and the integration controls just passed in.

Thus, an integrable object generates its two-state integrator that was discussed in section 4.3.1. Note that the integration-controls and time-integrator that the integrator-constructor creates are created elsewhere, in the generation of an integration group.

To extend this architecture, it is really only necessary to implement a different IntegrableObject (other than a Simple6DofDynBody), and in that implementation, put a different method in for the generation of the time-derivative of state[0]. The Simple6DofDynBody has a translational two-state, and a rotational two-state, with each two-state representing 3 degrees of freedom - hence 6 degrees of freedom total. The new object can, in principle, have any number of two-states, hence any number of degrees of freedom; the only requirement is that all of the two-state sets be mutually independent.

Chapter 5

Inspections, Tests, and Metrics

5.1 Inspection

Inspection Integration_1: Top-level Inspection

This document structure, the code, and associated files have been inspected, and together satisfy requirement Integration_1.

Inspection Integration_2: Mathematical Formulation

The implementations of the various integration techniques implement the corresponding algorithms as described in section 3.2.

By inspection, the Integration Model partially satisfies requirements Integration_3 and Integration_4.

Inspection Integration_3: Design Inspection

The model design uses a set of abstract classes. Each supported technique builds upon these abstract classes using a common scheme. This scheme of using derived classes enables extensions to the model, one of the key goals of (and requirements on) the model. A new model comprises extensions to three abstract classes. The TwoStateIntegrator class abstractly propagates generalized position and velocity. The TimeIntegrator class abstractly propagates time in concert with the state integrator. The IntegratorConstructor creates new instances of the state and time integrator.

By inspection, the Integration Model partially satisfies requirements Integration_2, Integration_5, Integration_6, Integration_7, and Integration_8.

5.2 Tests

This section describes various tests conducted to verify and validate that the Integration Model satisfies the requirements levied against it. All verification and validation test source code, simulations and procedures are archived in the JEOD directory `models/utils/integration/verif`.

Test Integration_1: Numerical Integration

Background The primary purpose of this test is to determine whether the Integration Model integration techniques do indeed integrate states over time. As every integration technique is subject to error, a secondary purpose of this test is to assess the accuracy of the various integration techniques.

Several complicating factors arise when testing numerical propagation techniques. Three such factors are

- Even the best numerical propagation will fail to provide accurate results when the characteristic frequency of the system being propagated approaches to the integration frequency.
- Even the best numerical propagation will fail to provide accurate results when the integration frequency is much smaller than the characteristic frequency of the system being propagated.
- Numerical errors tend to grow with time. With many integrators this error growth is non-linear. Integrate for too long a time and the result will be meaningless.

For a given problem, the integrated state will track the true state to within a reasonable degree of accuracy if the integration frequency is held to within some limited frequency range and if the integration interval is not too long. This test characterizes the various techniques supplied with JEOD to assess the integration frequency and interval over which the techniques can be trusted.

Test description This test involves using the numerical integration techniques to propagate five sample problems that have analytic solutions. Three of these sample problems demonstrate the propagation of a body rotating in three dimensional space using quaternions as the generalized position and angular velocities as the generalized velocity. The other two sample problems demonstrate the propagation of a body translating in three dimensional space, this time using Cartesian position and velocity as the generalized position and velocity. Note that these are the representations of rotational and translational state used throughout in JEOD. The test articles are

A freely rotating (torque-free) spherical body A torque-free body with a spherical mass distribution has very simple dynamics. The body rotates at a constant angular rate about a fixed axis.

A torque-free symmetric top Making the body have a single axis of symmetry adds the complicating factor of inertial torque. This fictitious torque is a consequence of using a non-inertial reference frame for the representation of rotational state.

A simple rotational harmonic oscillator In this problem an external torque that is proportional to the angular displacement of the body from some reference orientation is applied to the body. The result is a simple harmonic oscillator.

A body following a Keplerian orbit This is the first of the two translational state problems. In this problem the body is subject to a central inverse square force with the central object located at the origin of the integration frame. The result is a Keplerian orbit about the origin. The test harness for this problem can handle any closed orbit. Circular orbits are used in the tests below.

A spring-mass-damper system This final problem is that of a damped harmonic oscillator. The test harness can handle any kind of damping (under-damped, critically damped, and over-damped). A lightly damped oscillator is used in the tests below.

In a sense these are overly simplistic problems, particular in their canonical forms. The integration can be made a bit tougher by rotating away from the canonical description. The test harness uses this approach. This rotation from canonical adds a small amount of initial error to the system.

This test involves Monte-Carlo testing over a wide range of integration frequencies and a large number of random initial conditions. Because of the large number of test cases involved this test examines the behaviors of the various integration techniques from a statistical point of view. This test involves determining "three-sigma" error bounds as a function of the sample problem, the integration technique, the integration frequency, and the run duration. These assessments are made using order statistics.

Over a million test cases were produced to generate the results described below. Each run tests one set of the {sample problem, integration technique, integration frequency, run duration} problem space. The chosen integration technique is used to propagate the dynamical equations of motion for the chosen sample problem, with the integration occurring at the chosen frequency, and the simulation running for the chosen amount of time. Deviations between the propagated and computed states are assessed during the course of the run, with the worst-case deviation reported at the end of the run.

The integration frames for a given run are rotated from canonical by pseudorandom amounts. This random variation was repeated 369 times (370 is the minimum number of trials needed to assess the three-sigma bound via interpolation on order statistics) for each element of the problem space. After making the requisite number of runs, order statistics were used to estimate the three-sigma error bounds.

Test directory `SIM_integ_test`
The `S_define` uses the test scenario derivative and evaluation functions to propagate and integrate the selected states.

Test scripts `run_cases.pl` and `scan_runs.pl`
The first script runs the simulation many, many times. The second script analyzes the output to estimate the three-sigma error bounds and generates the plots displayed below.

Success criteria Any given integration technique will perform well only for a limited span of time and only if the integration is performed at some reasonable frequency. This presents a challenge in testing. An overly strict set of criteria will result in all techniques deemed to have failed. Instead, the performance is to be categorized as follows.

The rotation test cases are deemed to perform

- Exceptionally if the three-sigma angular error bounds do not exceed 10^{-9} degrees,
- Well if the three-sigma angular error bounds do not exceed 10^{-6} degrees,
- Acceptably if the three-sigma angular error bounds do not exceed 10^{-3} degrees,
- Marginally if the three-sigma angular error bounds do not exceed 1 degree, and
- Unacceptably if the three-sigma angular error bounds exceed 1 degree.

The translation tests have similar error classifications:

- Exceptionally if the three-sigma relative position error bounds do not exceed 10^{-12},
- Well if the three-sigma relative position error bounds do not exceed 10^{-9},
- Acceptably if the three-sigma relative position error bounds do not exceed 10^{-6},
- Marginally if the three-sigma relative position error bounds do not exceed 10^{-3}, and
- Unacceptably if the three-sigma relative position error bounds exceed 10^{-3}.

Test results The three-sigma error bounds are portrayed as log-log graphs in figures 5.1 to 5.5.

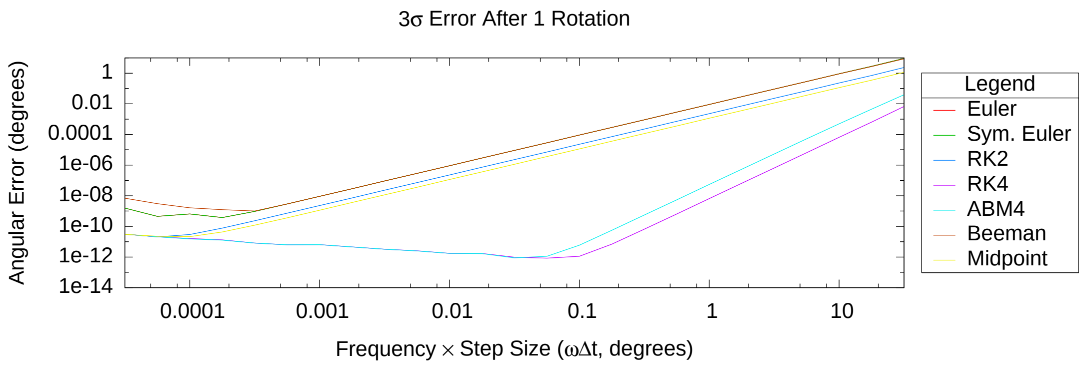
A synopsis of the results:

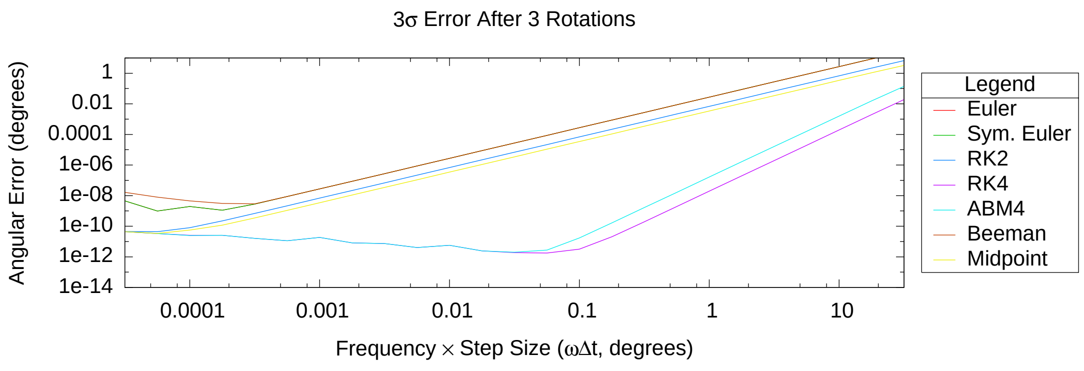
- Only the two fourth-order techniques, RK4 and ABM4, yield exceptional performance. The optimal integration rate for these two techniques is $\omega\Delta t \approx 0.05$ degrees for the investigated non-stiff problems, or about 1 Hz for a vehicle in low Earth orbit.
- None of the integration techniques perform exceptionally at very small ($\omega\Delta t < 10^{-4}$ degrees) time steps, which is about 650 Hz for a vehicle in low Earth orbit.
- All of the integration techniques break down as the integration frequency approaches the Nyquist frequency ($\omega\Delta t = 180$ degrees). Where this breakdown occurs depends on the problem being solved and on the integration technique being used to solve it. All of the techniques provided with JEOD begin performing poorly well below the Nyquist frequency.
- Most of the integrators perform acceptably or better at some intermediate integration frequency and for some limited amount of time. The two marked exceptions are the Euler integrators. The basic Euler technique fares quite poorly at all tested rates. The symplectic Euler technique begins to perform marginally (performance that might pass for acceptable accuracy in some circumstances) at extremely fast integration frequencies.

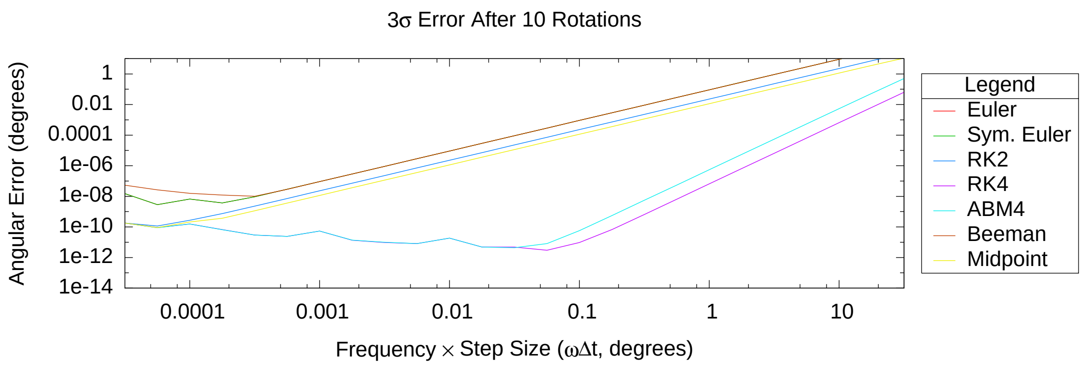
In summary, basic Euler technique fails to meet any reasonable expectation of accuracy requirements. The symplectic Euler technique performs marginally. All of the other techniques meet accuracy requirements over a wide range of integration frequencies. All of the techniques of course fail when the integration frequency is too low. That failure is the nature of integrating at a frequency too close to the Nyquist frequency.

Addendum to Test Results:

Comparison of the default configuration of the Gauss-Jackson method (8th order, no bootstrapping, no convergence testing) against the ABM4 and RK4 methods was conducted for the orbit propagation test for a range of integration rates.







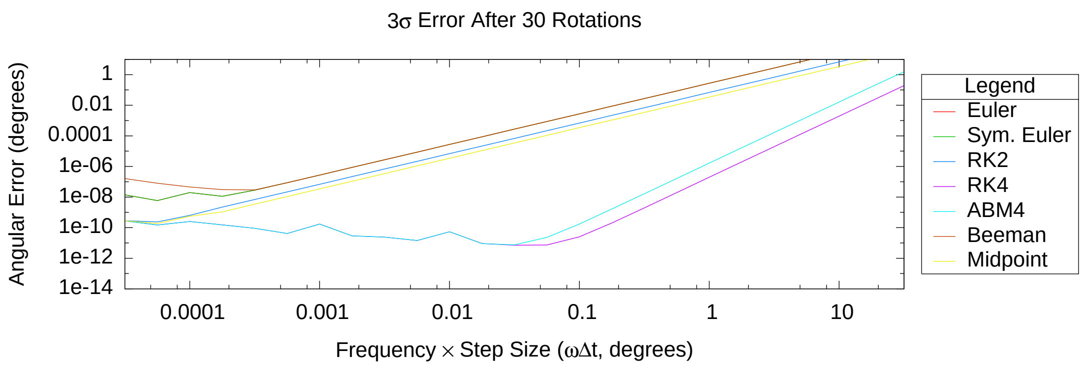
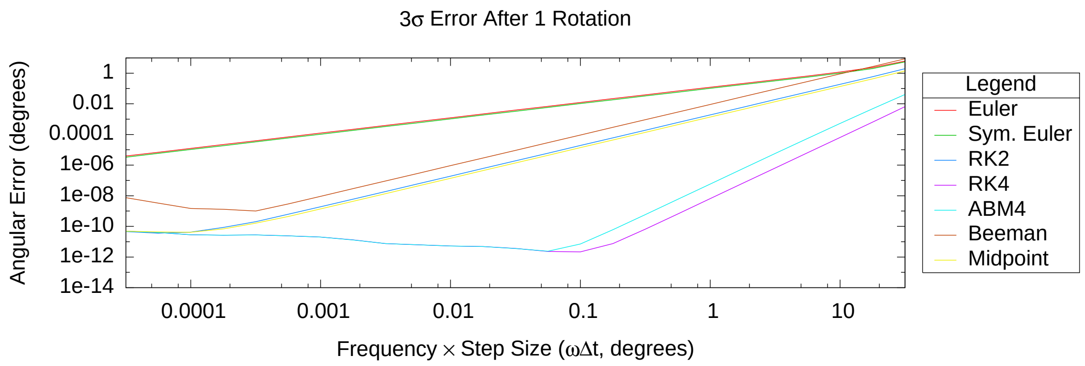
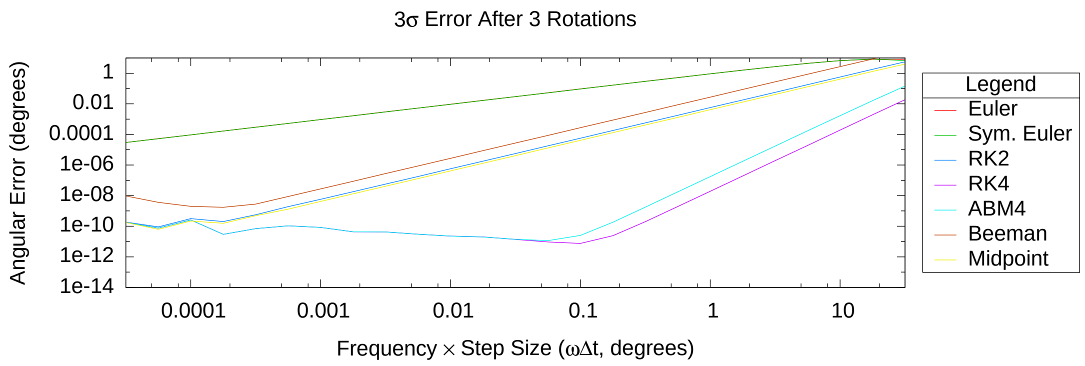
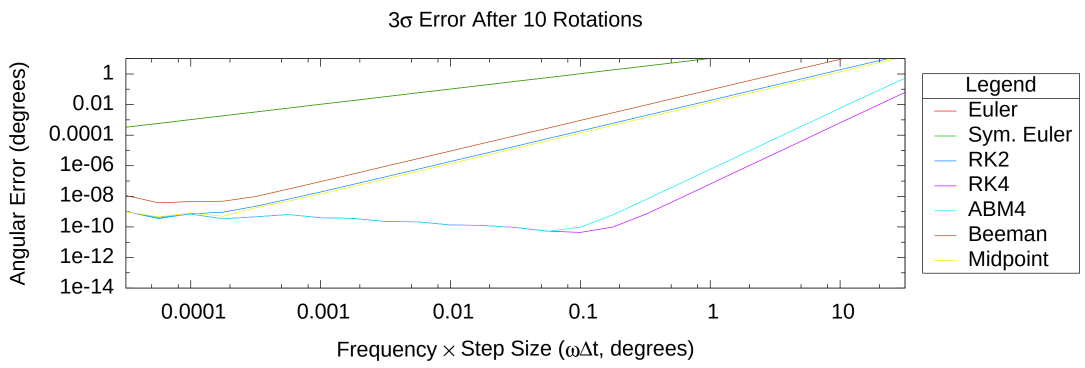


Figure 5.1: Torque-Free Sphere 3σ Errors







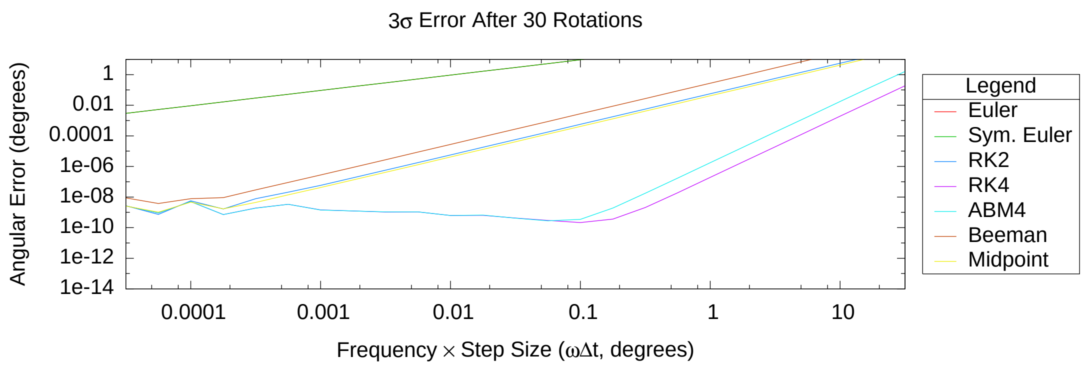
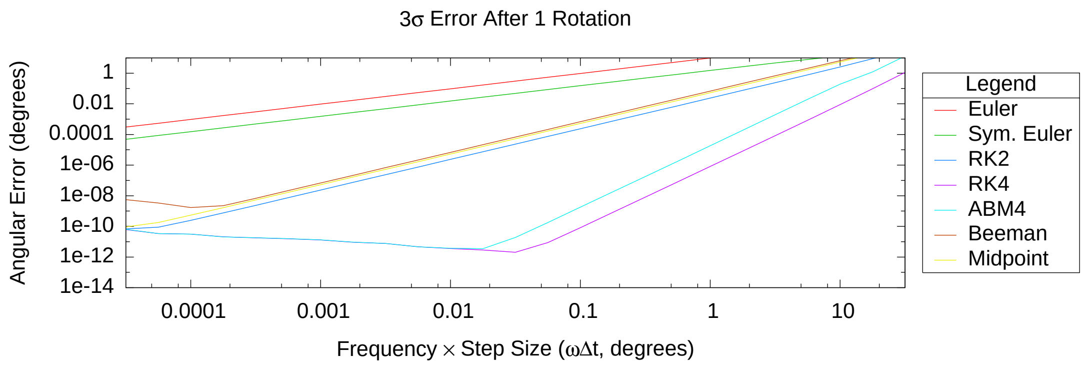
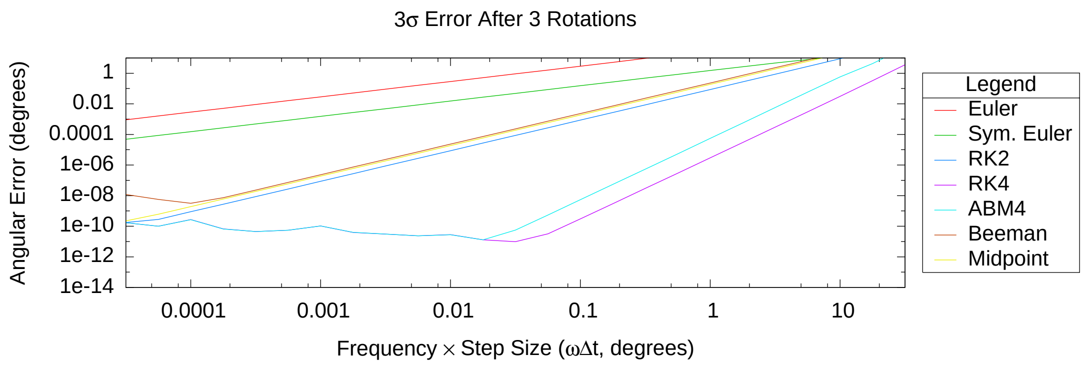
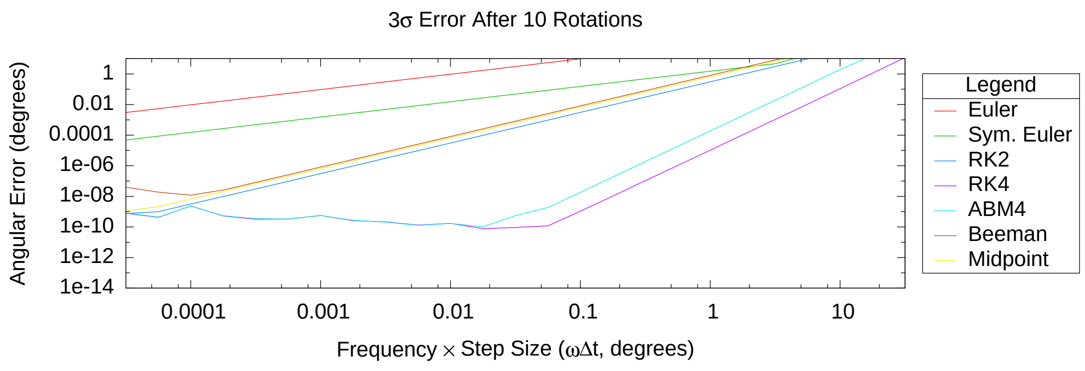


Figure 5.2: Torque-Free Symmetric Top 3σ Errors







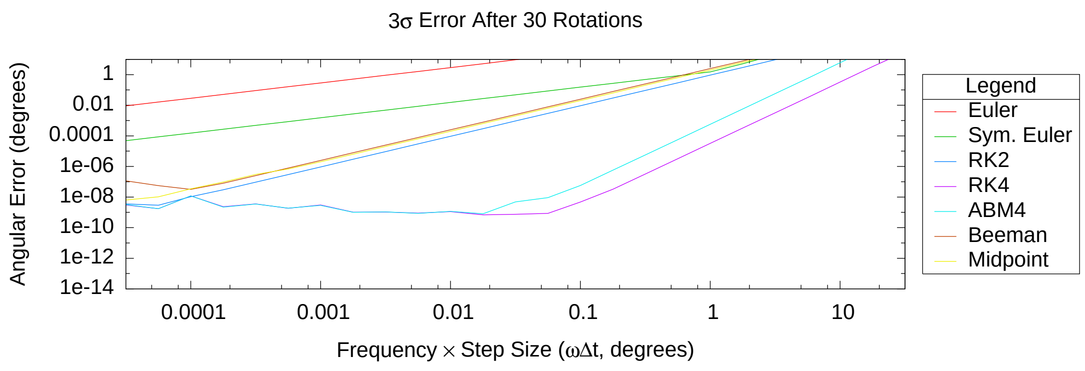
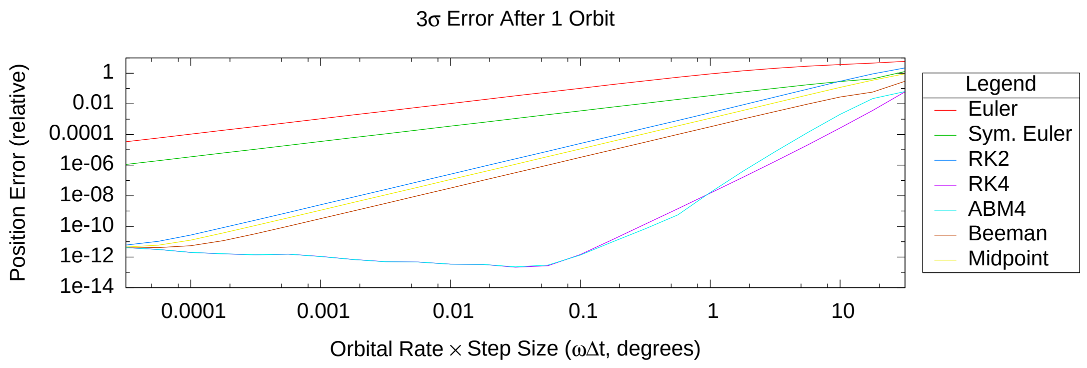
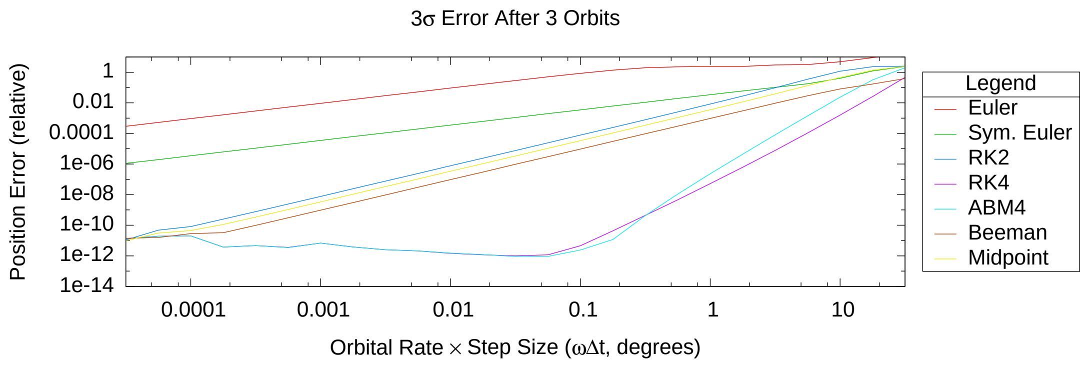
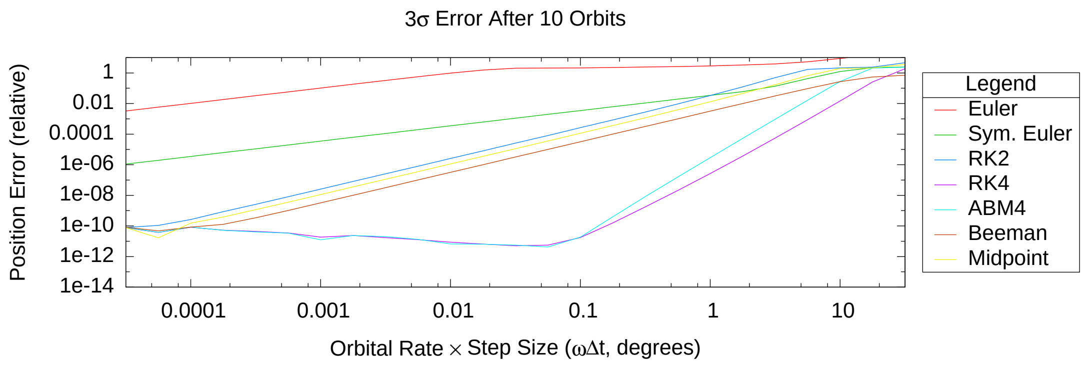


Figure 5.3: Spherical Harmonic Oscillator 3σ Errors







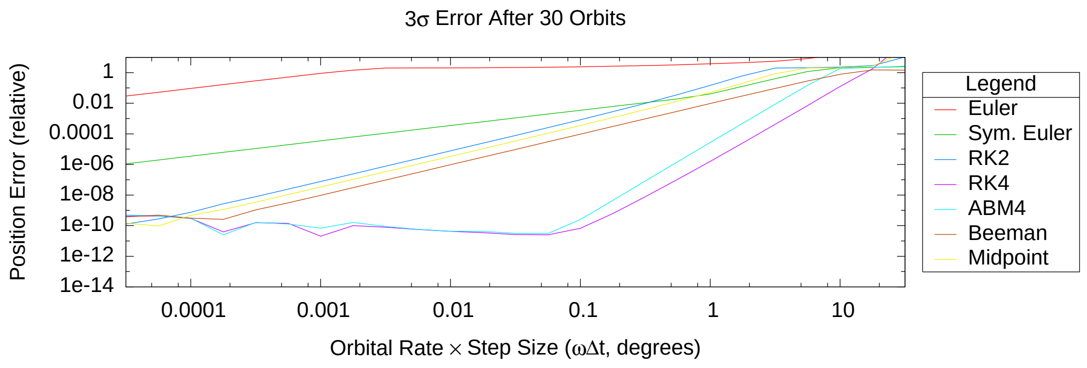
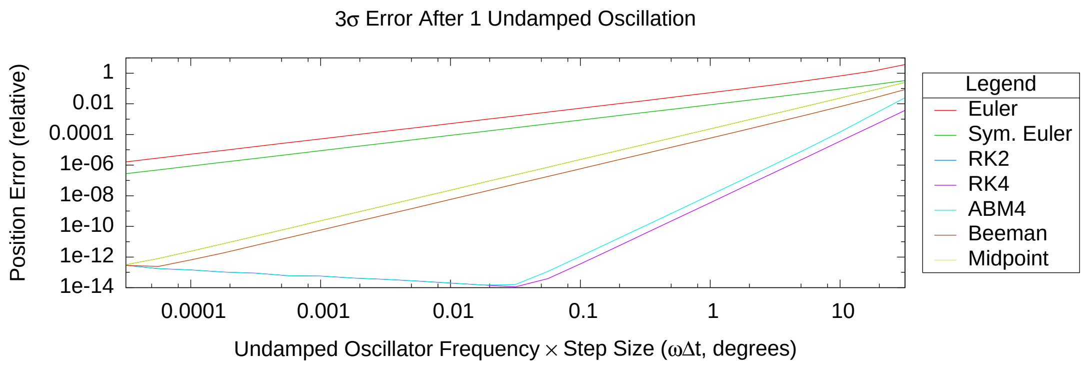
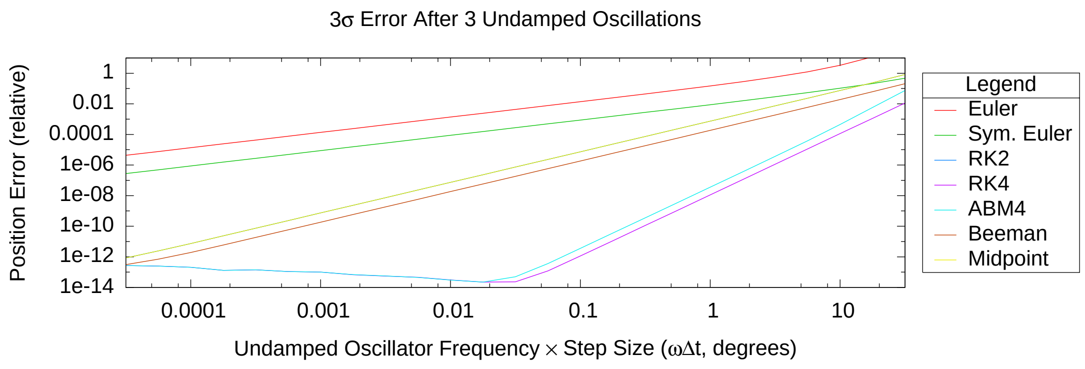
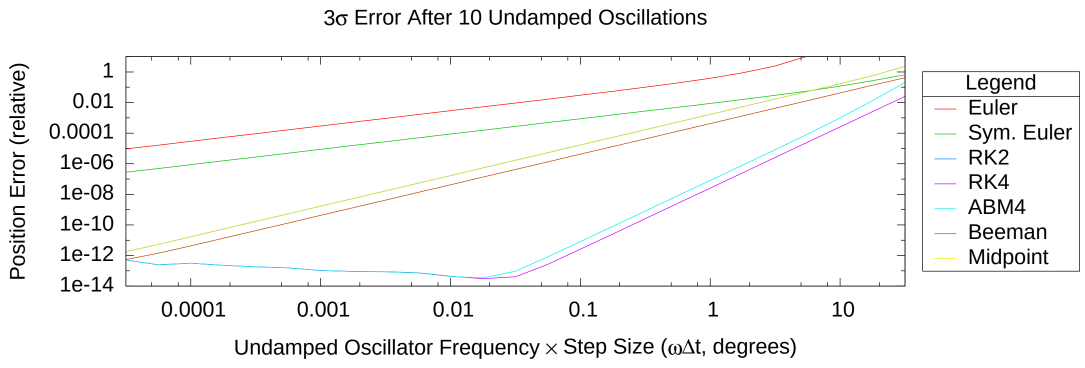


Figure 5.4: Circular Orbit 3σ Errors







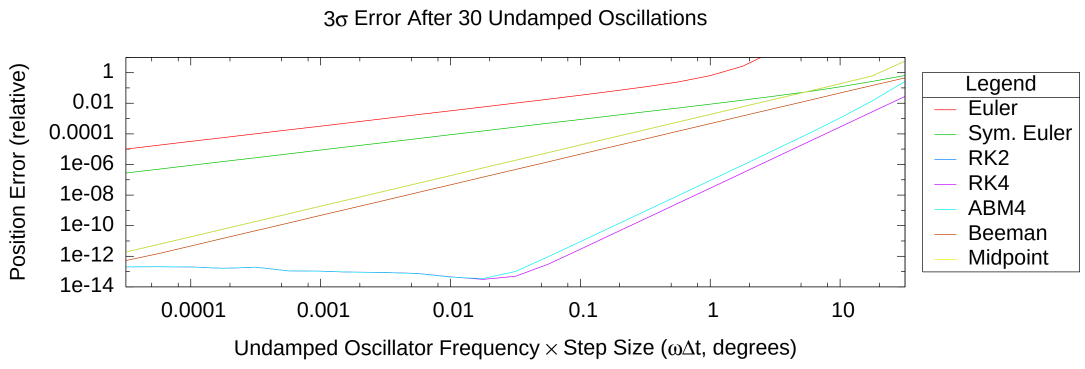


Figure 5.5: Spring-Damper 3σ Errors

Preliminary results indicate:

- **High rates** ($\omega \Delta t \sim O(10^-4 - 10^{-3}) deg.$):
  Surprisingly, the decrease in performance seen in ABM4 and RK4 is not nearly so pronounced in Gauss-Jackson. Consequently, Gauss-Jackson showed marginally superior performance by one to two orders of magnitude over this range. These tests were run over a 5-orbit period.
- **Mid-range rates** ($\omega \Delta t \sim O(10^-2 - 10^{-1}) deg.$):
  Over this range, the performance stays flat, perhaps diminishing slightly to be comparable to that of ABM4 and RK4. These tests were run over a 50-orbit period.
- **Low rates** ($\omega \Delta t \sim O(10^0 - 10^1) deg.$):
  The performance continues to diminish at longer time-steps, but much more slowly than the ABM4 and RK4. By $\omega \Delta t \approx 1 deg.$, the Gauss-Jackson algorithm performs three to four orders of magnitude better than the ABM4 and RK4 algorithms. This trend continues until the ABM4 starts to lose stability at around 10 degrees, at which point the Gauss-Jackson performance falls to marginal. By 30 degrees, RK4 has also lost stability, and while Gauss-Jackson continues to propagate nicely, the accumulated errors are reaching around 10% after 50 orbits.

The Gauss-Jackson algorithm has a number of user-definable parameters that control its behavior. A description of these is found in the User Guide (see *Gauss-Jackson Parameters* (on page 27)), and verification of their functionality is found in Test Integration_2.

Applicable Requirements This test completes the satisfaction of the requirements Integration_2 and Integration_3, and partially satisfies the requirements Integration_5 and Integration_6.

Test Integration_2: Gauss-Jackson Parameters

Background The Gauss-Jackson algorithm has a number of user-definable parameters that control its behavior. A description of these is found in the User Guide (see *Gauss-Jackson Parameters* (on page 27)). This test verifies their functionality, and provides a quantitative assessment of their respective behaviors for propagation in the very simple case of a circular orbit in a spherical gravity field.

Test directory `SIM_GJ_test`

Test description A number of runs were made to perform the following tests:

1. A comparison of Gauss-Jackson against RK4 for a range of time-steps.
2. A study of the effect of adding the evaluation-step after the corrector-step, and altering the convergence threshold.
3. A study of the effect of adding the bootstrap method, for both short time-step and long time-step simulations.
4. A study of the extent to which the order of the integrator affects its accuracy.

All runs were performed on a very simple orbital case, involving a vehicle in a circular orbit, of period around 7000 seconds, around an earth-like planet with spherical gravity. The only comparison investigated was the magnitude of the position vector against the reference orbital radius. The simplicity of this test must be considered when evaluating the results presented here; the validity of applying these data to more complex environments is inherently questionable. Nevertheless, these data provide a starting point for identifying optimal integrator configuration.

Success criteria The fundamental test is to ensure that the simulation will run with a diverse array of integrator parameters. The secondary test is to quantify the effect of changing those parameters.

Test results The fundamental test passes, there was no configuration found for which the simulation failed.

The quantitative comparison of the different parameters follows.

Gauss-Jackson versus RK4

The Gauss-Jackson integrator used to compare against RK4 uses the default configuration - 8th order, no evaluation step, no bootstrapping. Integration step sizes are tested in powers of 10; in Figure 5.6, the legend indicates the step-size:

- step00 - 0.01 seconds
- step0 - 0.1 seconds
- step1 - 1 second
- step10 - 10 seconds
- step 100 - 100 seconds

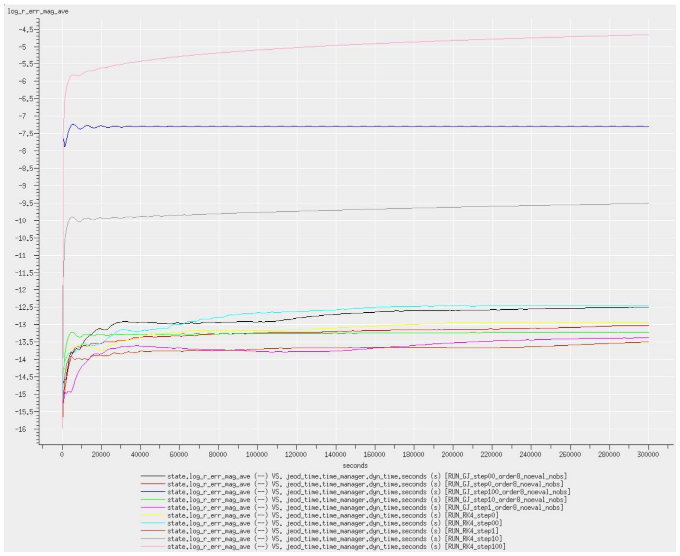


Figure 5.6: The variation with time of the log of the averaged (over the simulation to time, t) absolute value of the relative difference between the magnitude of the numerically integrated position vector and the radius of the circular orbit that the vehicle follows; $log_{10}(<\frac{|\vec{r}|-r_{circ}}{r_{circ}}>) \sim t$. Data are taken from Gauss-Jackson and RK4 integrators with a selection of integration rates.

Three runs stand out as being particularly weak. As expected, the RK4 algorithm performs poorly at 10-second and 100-second cycle-times (corresponding to $\omega\Delta t \sim 0.5, 5deg$). The Gauss-Jackson algorithm also struggles at the 100-second step, but this is not particularly surprising since it is initialized with a RK4 integrator running at 100 seconds. See the bootstrapping comparison below for details on circumventing this problem.

Of the remaining runs, the two algorithms running at 0.01 seconds fare relatively poorly. These five runs are removed for comparison of the variation with time of the position error, $\frac{|\vec{r}|-r_{circ}}{r_{circ}}$ (where r_{circ} is the reference orbital radius), shown in Figure 5.7.

The results indicate limited differential between the algorithms at the time-steps selected for Figure 5.7, i.e., 0.1 seconds to 1.0 seconds for RK4; 0.1 seconds to 10.0 seconds for GJ.

Significance of the Evaluate Step in Predict-Correct-Evaluate

For this illustration, baseline GJ integrators (8th order, no evaluation step, no bootstrapping) with 1-second and 100-second integration steps were compared against similar GJ integrators with a convergence criterion applied to them, specifying that the state must converge to within 1 part in 10^{15}, approximately the limit of the *double* data specification. Such a tight requirement was imposed because of the simplicity and predictability of the simulation; the integrator should be able to get very close to the converged solution with only one correction phase.

Indeed, even with such a tight requirement, there was no discernible difference in the results from the 1-second integrators. At the 100-second integration rate, a very small difference (about 1 part in 10^4) was observed by the end of the simulation in the value $\frac{|\vec{r}|-r_{circ}}{r_{circ}}$. With the value $\frac{|\vec{r}|-r_{circ}}{r_{circ}} \sim O(10^{-7})$, this indicates a difference in the magnitude of the position, resulting from the performance of the evaluate step, of about 1 part in 10^{11}.

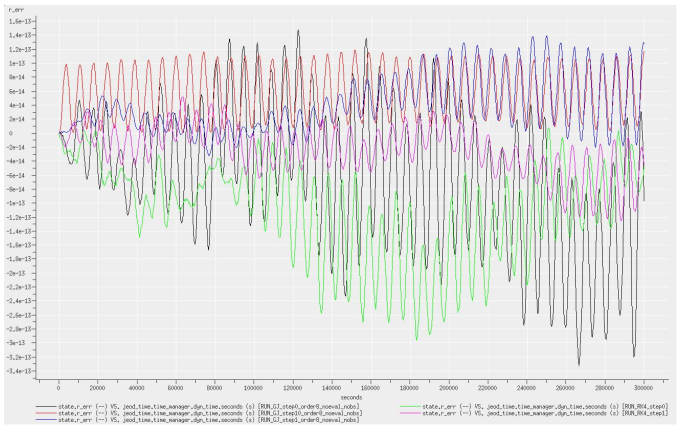


Figure 5.7: The variation with time of the relative difference between the magnitude of the numerically integrated position vector and the radius of the circular orbit that the vehicle follows; $\frac{|\vec{r}|-r_{circ}}{r_{circ}} \sim t$. Data are taken from Gauss-Jackson and RK4 integrators with a selection of integration rates.

Effect of Adding the Bootstrap Method

The bootstrap method takes a specified value for the maximum RK4 time-step that can be used in the GJ integrator priming phase, and gradually ramps that time-step up until the GJ integrator is running with a cycle-time equal to the desired tour time. The bootstrap method should be most useful when the GJ tour time is in the regime where RK4 performs poorly.

First, an erroneous application was tested to ensure that mis-application does not cause undesirable behavior. The 1-second integrator was bootstrapped with a 0.1-second RK4 integrator, and then with a 0.01-second RK4 integrator (note that at 1-second, RK4 is near its optimal performance, so bootstrapping is not needed in this scenario, and bootstrapping with a sub-optimal time-step is clearly a detrimental process). As shown in Figures 5.8 and 5.9, the effect in this regime is minimal; indeed the non-bootstrapped simulation is indistinguishable from the one bootstrapped at 0.1-seconds; both of these marginally outperform the third simulation.

Next, the correct application of the bootstrap was evaluated, bootstrapping the 100-second integrator off a 1-second (i.e. near-optimal) RK4 priming integrator. The effect is quite pronounced, producing an improvement of around 5 orders of magnitude, shown in Figure 5.10. This improvement makes the 100-second GJ integrator a reasonable tool, still slightly off-optimal for numerical precision, but with significantly improved speed in the situation where the environment takes significant computation. (Note - a period of 100-seconds corresponds to an angular period of approximately 5 degrees).

Further testing indicates that, while the bootstrap algorithm significantly enhances performance, it is insufficient to overcome the rapid degradation of performance for angular rates greater than about 5 degrees per step. As illustrative points, at 15 degrees the performance is back to the 10^{-7} range that we had a 5-degrees without bootstrapping; by 30 degrees, the performance is worse, even with bootstrapping, than that of RK4 at 5-degrees; by 40 degrees, the algorithm becomes completely unreliable.

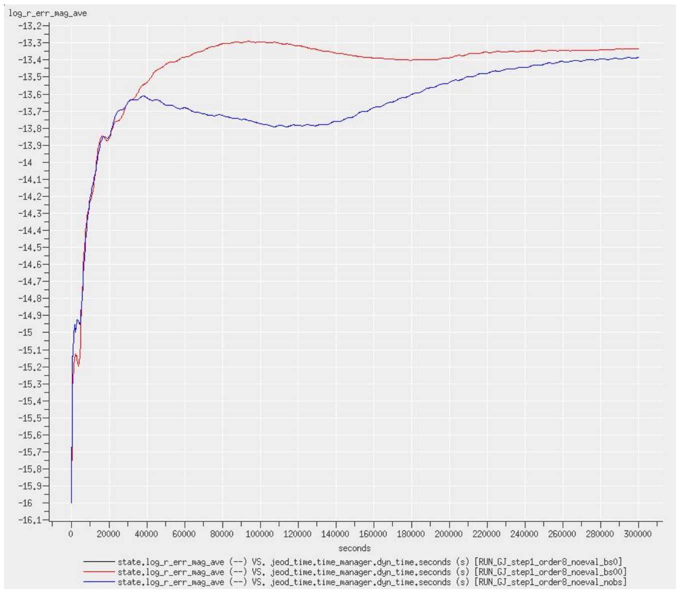


Figure 5.8: The variation with time of the log of the averaged (over the simulation to time, t) absolute value of the relative difference between the magnitude of the numerically integrated position vector and the radius of the circular orbit that the vehicle follows; $log_{10}(<\frac{|\vec{r}|-r_{circ}}{r_{circ}}>) \sim t$. Data are taken from Gauss-Jackson integrators with 1-second rates, with and without bootstrapping.

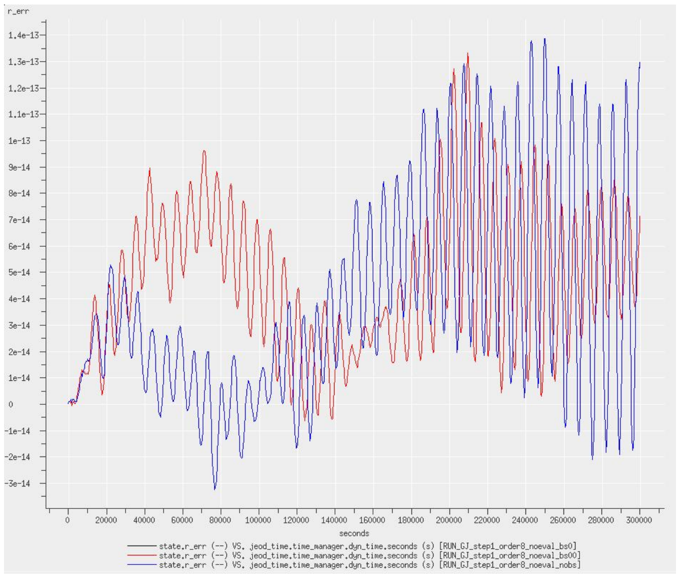


Figure 5.9: The variation with time of the relative difference between the magnitude of the numerically integrated position vector and the radius of the circular orbit that the vehicle follows; $\frac{|\vec{r}|-r_{circ}}{r_{circ}} \sim t$. Data are taken from Gauss-Jackson integrators with 1-second rates, with and without bootstrapping.

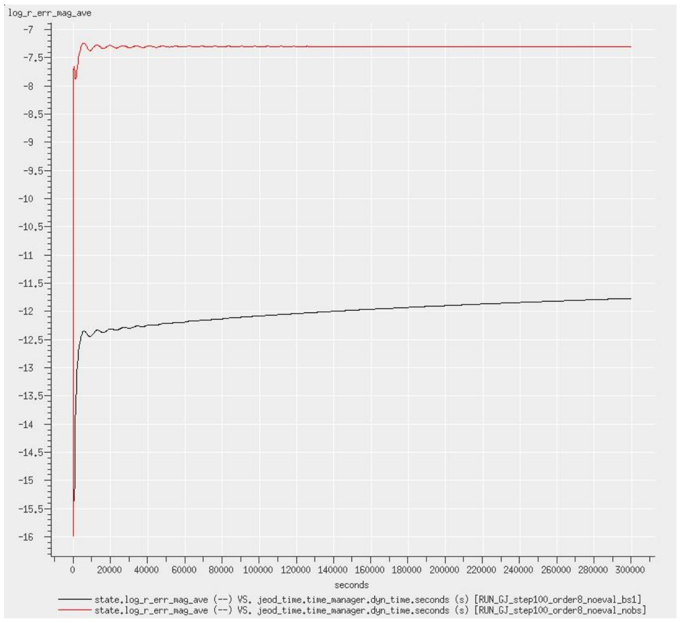


Figure 5.10: The variation with time of the log of the averaged (over the simulation to time, t) absolute value of the relative difference between the magnitude of the numerically integrated position vector and the radius of the circular orbit that the vehicle follows; $log_{10}(< \frac{|\vec{r}|-r_{circ}}{r_{circ}} >) \sim t$. Data are taken from Gauss-Jackson integrators with 100-second rates, with and without bootstrapping.

Effect of Changing the Order of the Integrator

For such a simple scenario, it is not expected that the order of the integrator should have a significant effect, and that is what is found. Figure 5.11 illustrates the performance for integrators with order 2, 4, 8, 12, and 16. While the 16th order provides the best performance, it is a marginal result, and is followed by the 2nd order. No conclusion can be drawn about the effect of changing the order of the integrator from this scenario.

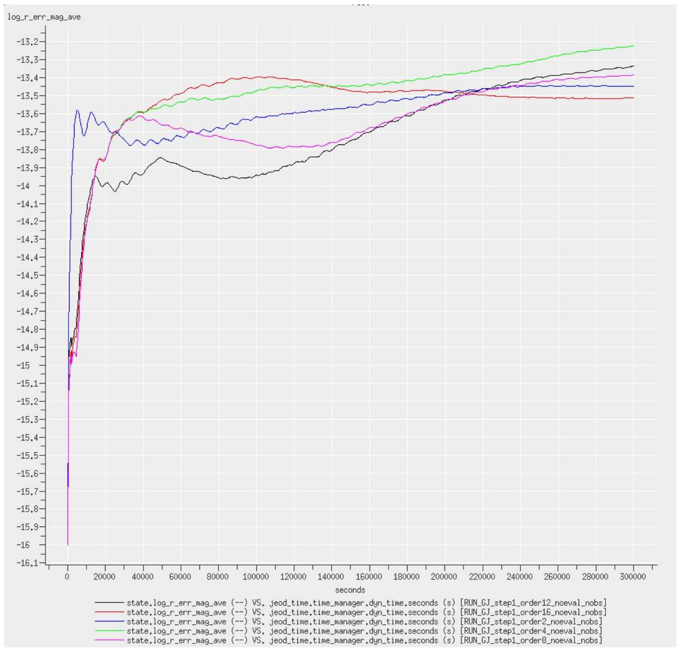


Figure 5.11: The variation with time of the log of the averaged (over the simulation to time, t) absolute value of the relative difference between the magnitude of the numerically integrated position vector and the radius of the circular orbit that the vehicle follows; $log_{10}(< \frac{|\vec{r}| - r_{circ}}{r_{circ}} >) \sim t$. Data are taken from Gauss-Jackson integrators with 1-second rates, with varying order.

Applicable Requirements This test completes the satisfaction of requirements Integration_4 and Integration_6.

Test Integration_3: Multiple Integrators

Background The purpose of this test is to verify that the model can accommodate an arbitrary number of state integrators. The ability of the Integration Model to accommodate multiple integrators was implicitly demonstrated in test Integration_1. The large number of individual test cases needed by that test made it highly advantageous to instrument the test harness code with the ability to test multiple integrators in parallel. This test explicitly demonstrates that capability.

Test directory `SIM_integ_test`
The same simulation is used for all of the Integration Model verification tests.

Test scripts `run_cases.pl` and `test_multi.tcsh`
The latter script drives the former script, which is the same script used to drive the simulation for test Integration_1.

Test description The `test_multi.tcsh` script is used to generate what should be two identical sets of output data. The script is used to run the simulation a number of times in series for one set (N simulation runs of one case each) and in parallel for the other set (one simulation run of N test cases). The test is repeated for each integration technique provided by JEOD.

Success criteria A test of an individual technique succeeds if the serial and parallel runs generate identical output. The test succeeds as a whole if each of the individual tests succeed.

Test results All tests pass.

Applicable Requirements This test partially satisfies requirements Integration_7 and Integration_8.

Test Integration_4: Model Extensibility

Background The purpose of this test is to assess whether a numerical integration technique that is not supported by JEOD can be created and used in a simulation.

Test description The second order midpoint method is not supported in JEOD. This test involves implementing this method as a demonstration of the extensibility of the model.

Success criteria The test succeeds if the method can be implemented within the confines of the Integration Model architecture and successfully used in lieu of the supported methods in the various tests of algorithm accuracy.

Test results The new integration model was successfully implemented and tested. The results from using this integration technique are presented in test Integration_1.

Applicable Requirements This test completes the satisfaction of requirement Integration_5.

Test Integration_5: Multiple Integration Groups

Background The purpose of this test is to verify that a simulation can operate with multiple integrators operating simultaneously with different conditions.

Test Description The simulation SIM_prop_planet_T10 utilizes two different integration rates for propagation of planets (propagating their state, rather than using the ephemeris model). At one point in the simulation, one of the bodies is switched from one integration group to the other.

Success Criteria The simulation needs to demonstrate:

1. Both groups can propagate independently at different rates.
2. A body can successfuly transition from one group to the other.

Test Results Both test criteria were satisfied.

Applicable Requirements This test completes the satisfaction of requirements Integration_7 and Integration_8.

5.3 Regression Tests

Only the Gauss-Jackson test *SIM_GJ_test* is in a form that is suitable for regression testing.

The other tests use input files that are created in temporary directories and deleted after use. None of these tests use logged data.

Several regression tests were created to address this issue. The regression tests are not used to verify and validate the model. They are used instead to determine whether something went awry during nightly builds and prior to release. The regression tests follow the standard JEOD naming convention, with input files in subdirectories within the SET_test directory and comparison data in in corresponding subdirectories in the SET_test_val directory.

5.4 Requirements Traceability

Table 5.1 summarizes the inspections and tests that demonstrate the satisfaction of the requirements levied on the model.

Table 5.1: Requirements Traceability

Requirement	**Traces to**
Integration_1 Project Requirements	Insp. Integration_1 Top-level Inspection
Integration_2 ER7 Utilities Framework	Insp. Integration_3 Design Inspection
	Test Integration_1 Numerical Integration
Integration_3 ER7 Utilities Techniques	Insp. Integration_2 Mathematical Formulation
	Test Integration_1 Numerical Integration
Integration_4 Long-Arc Integration	Insp. Integration_2 Mathematical Formulation
	Test Integration_2 Gauss-Jackson Parameters
Integration_5 Extensibility	Insp. Integration_3 Design Inspection
	Test Integration_1 Numerical Integration
	Test Integration_4 Model Extensibility
Integration_6 Support JEOD ODEs	Insp. Integration_3 Design Inspection
	Test Integration_1 Numerical Integration
	Test Integration_2 Gauss-Jackson Parameters
Integration_7 Multiple States	Insp. Integration_3 Design Inspection
	Test Integration_3 Multiple Integrators
	Test Integration_5 Multiple Integration Groups
Integration_8 Multiple Integrators	Insp. Integration_3 Design Inspection
	Test Integration_3 Multiple Integrators
	Test Integration_5 Multiple Integration Groups

5.5 Metrics

Table 5.2 presents coarse metrics on the source files that comprise the model.

Table 5.2: Coarse Metrics

	Number of Lines			
File Name	**Blank**	**Comment**	**Code**	**Total**
Total	0	0	0	0

Table 5.3 presents the extended cyclomatic complexity (ECC) of the methods defined in the model.

Table 5.3: Cyclomatic Complexity

Method	**File**	**Line**	**ECC**
jeod::GaussJackson CoefficientsPair::Gauss JacksonCoefficientsPair ()	gauss_jackson/include/gauss_jackson_coefficients_pair.hh	106	1
jeod::GaussJackson CoefficientsPair::~Gauss JacksonCoefficientsPair ()	gauss_jackson/include/gauss_jackson_coefficients_pair.hh	115	1
jeod::GaussJackson CoefficientsPair::configure (int max_order)	gauss_jackson/include/gauss_jackson_coefficients_pair.hh	123	1
jeod::GaussJacksonCoeffs:: GaussJacksonCoeffs ()	gauss_jackson/include/gauss_jackson_coeffs.hh	110	1
jeod::GaussJacksonCoeffs:: GaussJacksonCoeffs (const GaussJacksonCoeffs & src)	gauss_jackson/include/gauss_jackson_coeffs.hh	121	3
jeod::GaussJacksonCoeffs:: operator= (GaussJackson Coeffs src)	gauss_jackson/include/gauss_jackson_coeffs.hh	143	1
jeod::GaussJacksonFirstOrder ODEIntegrator::er7_utils:: GaussJacksonFirstOrderOD EIntegrator (const er7_utils::IntegratorConstructor & priming_constructor, GaussJacksonIntegration Controls & controls, unsigned int size_in, er7_utils::IntegrationControls & priming_controls)	gauss_jackson/include/gauss_jackson_first_order_ode_integrator.hh	101	1
jeod::GaussJacksonFirstOrder ODEIntegrator::er7_utils:: GaussJacksonFirstOrderOD EIntegrator (const Gauss JacksonFirstOrderODE Integrator & src)	gauss_jackson/include/gauss_jackson_first_order_ode_integrator.hh	123	1
jeod::GaussJacksonFirstOrder ODEIntegrator::~Gauss JacksonFirstOrderODE Integrator ()	gauss_jackson/include/gauss_jackson_first_order_ode_integrator.hh	134	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::GaussJacksonFirstOrderODEIntegrator::operator= (GaussJacksonFirstOrderODEIntegrator src)	gauss_jackson/include/gauss_jackson_first_order_ode_integrator.hh	140	1
jeod::GaussJacksonFirstOrderODEIntegrator::FirstOrderODEIntegrator::swap (GaussJacksonFirstOrderODEIntegrator & other)	gauss_jackson/include/gauss_jackson_first_order_ode_integrator.hh	150	1
jeod::GaussJacksonFirstOrderODEIntegrator::er7_utils::create_copy ()	gauss_jackson/include/gauss_jackson_first_order_ode_integrator.hh	160	1
jeod::GaussJacksonFirstOrderODEIntegrator::reset_integrator ()	gauss_jackson/include/gauss_jackson_first_order_ode_integrator.hh	168	1
jeod::GaussJacksonFirstOrderODEIntegrator::er7_utils::integrate (double dyn_dt, unsigned int target_stage, double const * ER7_UTILS_RESTRICT deriv, double * ER7_UTILS_RESTRICT state)	gauss_jackson/include/gauss_jackson_first_order_ode_integrator.hh	176	1
jeod::GaussJacksonGeneralizedDerivSecondOrderODEIntegrator::operator= (GaussJacksonGeneralizedDerivSecondOrderODEIntegrator src)	gauss_jackson/include/gauss_jackson_generalized_second_order_ode_integrator.hh	140	1
jeod::GaussJacksonGeneralizedDerivSecondOrderODEIntegrator::reset_integrator ()	gauss_jackson/include/gauss_jackson_generalized_second_order_ode_integrator.hh	166	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJackson GeneralizedDerivSecond OrderODEIntegrator::er7_ utils::integrate (double dyn_ dt, unsigned int target_ stage, double const * ER7_ UTILS_RESTRICT acc, double * ER7_UTILS_RES TRICT vel, double * ER7_ UTILS_RESTRICT pos)	gauss_jackson/include/gauss_ jackson_generalized_second_ order_ode_integrator.hh	176	1
jeod::GaussJackson IntegrationControls:: operator= (GaussJackson IntegrationControls src)	gauss_jackson/include/gauss_ jackson_integration_ controls.hh	121	1
jeod::GaussJackson IntegrationControls::er7_ utils::get_priming_controls ()	gauss_jackson/include/gauss_ jackson_integration_ controls.hh	139	1
jeod::GaussJackson IntegrationControls::get_ coeff ()	gauss_jackson/include/gauss_ jackson_integration_ controls.hh	148	1
jeod::GaussJackson IntegrationControls::get_ config ()	gauss_jackson/include/gauss_ jackson_integration_ controls.hh	157	1
jeod::GaussJackson IntegrationControls::get_ state_machine ()	gauss_jackson/include/gauss_ jackson_integration_ controls.hh	166	1
jeod::GaussJacksonIntegrator Base::(GaussJacksonState Machine::GaussJackson IntegratorBase ()	gauss_jackson/include/gauss_ jackson_integrator_base.hh	193	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJacksonIntegratorBase::er7_utils::GaussJacksonIntegratorBase (const er7_utils::IntegratorConstructor & priming_constructor, const GaussJacksonIntegrationControls & controls, unsigned int size_in, er7_utils::IntegrationControls & priming_controls)	gauss_jackson/include/gauss_jackson_integrator_base.hh	223	1
jeod::GaussJacksonIntegratorBase::GaussJacksonIntegratorBase (const GaussJacksonIntegratorBase & src)	gauss_jackson/include/gauss_jackson_integrator_base.hh	276	1
jeod::GaussJacksonIntegratorBase::er7_utils::~GaussJacksonIntegratorBase ()	gauss_jackson/include/gauss_jackson_integrator_base.hh	314	1
jeod::GaussJacksonIntegratorBase::GaussJacksonStateMachine::base_reset ()	gauss_jackson/include/gauss_jackson_integrator_base.hh	330	1
jeod::GaussJacksonIntegratorBase::er7_utils::base_integrate (double dyn_dt, unsigned int target_stage, double const * ER7_UTILS_RESTRICT deriv, State state)	gauss_jackson/include/gauss_jackson_integrator_base.hh	340	10
jeod::GaussJacksonIntegratorBase::std::swap (GaussJacksonIntegratorBase & other)	gauss_jackson/include/gauss_jackson_integrator_base.hh	457	1
jeod::GaussJacksonIntegratorBase::GaussJacksonStateMachine::start_cycle (double dt, const double* ER7_UTILS_RESTRICT acc, const State & state)	gauss_jackson/include/gauss_jackson_integrator_base.hh	520	6

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJacksonIntegratorBase::edit_point (double dt, const double* ER7_UTILS_RESTRICT acc ER7_UTILS_UNUSED, State & state)	gauss_jackson/include/gauss_jackson_integrator_base.hh	572	1
jeod::GaussJacksonIntegratorBase::integrate_gj (double dt, unsigned int target_stage, int advance_index, int target_index, const double* ER7_UTILS_RESTRICT acc, const double* const * ahist, State & state)	gauss_jackson/include/gauss_jackson_integrator_base.hh	590	2
jeod::GaussJacksonIntegratorBase::downsample_hist ()	gauss_jackson/include/gauss_jackson_integrator_base.hh	626	1
jeod::GaussJacksonIntegratorBase::rotate_acc_hist ()	gauss_jackson/include/gauss_jackson_integrator_base.hh	638	1
jeod::er7_utils::create_primer (const er7_utils::IntegratorConstructor & priming_constructor, unsigned int size, er7_utils::IntegrationControls & priming_controls)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	93	1
jeod::er7_utils::replicate_primer (const er7_utils::FirstOrderODEIntegrator * src)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	110	2
jeod::er7_utils::integrate_primer (double dyn_dt, unsigned int target_stage, double const * deriv, GaussJacksonOneState & state)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	129	1
jeod::er7_utils::save_epoch_data (const double * acc, const GaussJacksonOneState & state)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	146	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::er7_utils::save_comparison_data (const GaussJacksonOneState & state, double * pos_hist_elem)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	163	1
jeod::er7_utils::initialize_edit_integration_constants (double dt)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	178	2
jeod::er7_utils::advance_edit_integration_constants (unsigned int index)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	196	2
jeod::er7_utils::initialize_predictor_integration_constants (double dt)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	215	2
jeod::er7_utils::advance_predictor_integration_constants (unsigned int index)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	233	2
jeod::er7_utils::mid_correct (unsigned int coeff_idx, double dt, GaussJackson OneState & state)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	252	2
jeod::er7_utils::predict (double dt, double const * const * ahist, GaussJacksonOne State & state)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	275	2
jeod::er7_utils::correct (double dt, const double* ER7_UTI LS_RESTRICT acc, Gauss JacksonOneState & state)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	298	2
jeod::er7_utils::test_for_convergence (const Gauss JacksonOneState & state, double* ER7_UTILS_REST RICT hist_data)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	322	4
jeod::er7_utils::swap_state (GaussJacksonOneState & item, GaussJacksonOne State & other_item)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	348	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::er7_utils::replicate_state (GaussJacksonOneState const & source, Gauss JacksonOneState & target)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	363	1
jeod::er7_utils::allocate_state_contents (GaussJacksonOne State & item)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	378	1
jeod::er7_utils::deallocate_state_contents (Gauss JacksonOneState & item)	gauss_jackson/include/gauss_jackson_integrator_base_first.hh	392	1
jeod::er7_utils::create_primer (const er7_utils::Integrator Constructor & priming_constructor, unsigned int size, er7_utils::Integration Controls & priming_controls)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	93	1
jeod::er7_utils::replicate_primer (const er7_utils::SecondOrderODEIntegrator * src)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	109	2
jeod::er7_utils::integrate_primer (double dyn_dt, unsigned int target_stage, double const * deriv, Gauss JacksonTwoState & state)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	127	1
jeod::er7_utils::save_epoch_data (const double* ER7_U TILS_RESTRICT acc, const GaussJacksonTwo State & state)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	145	1
jeod::er7_utils::save_comparison_data (const GaussJacksonTwoState & state, double* ER7_UTILS_RESTRICT pos_hist_elem)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	162	1
jeod::er7_utils::initialize_edit_integration_constants (double dt)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	176	2

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::er7_utils::advance_edit_integration_constants (unsigned int index)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	196	2
jeod::er7_utils::initialize_predictor_integration_constants (double dt)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	212	3
jeod::er7_utils::advance_predictor_integration_constants (unsigned int index)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	233	2
jeod::er7_utils::mid_correct (unsigned int coeff_idx, double dt, GaussJackson TwoState & state)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	249	2
jeod::er7_utils::predict (double dt, double const * const * E R7_UTILS_RESTRICT ahist, GaussJacksonTwo State & state)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	271	2
jeod::er7_utils::correct (double dt, const double* ER7_UTI LS_RESTRICT acc, Gauss JacksonTwoState & state)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	297	2
jeod::er7_utils::test_for_convergence (const Gauss JacksonTwoState & state, double* ER7_UTILS_REST RICT hist_data)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	329	4
jeod::er7_utils::swap_state (GaussJacksonTwoState & item, GaussJacksonTwo State & other_item)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	353	1
jeod::er7_utils::replicate_state (GaussJacksonTwoState const & source, Gauss JacksonTwoState & target)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	368	1
jeod::er7_utils::allocate_state_contents (GaussJacksonTwo State & item)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	383	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::er7_utils::deallocate_state_contents (GaussJacksonTwoState & item)	gauss_jackson/include/gauss_jackson_integrator_base_second.hh	397	1
jeod::GaussJacksonIntegratorConstructor::operator= (GaussJacksonIntegratorConstructor src)	gauss_jackson/include/gauss_jackson_integrator_constructor.hh	116	1
jeod::GaussJacksonIntegratorConstructor::get_class_name (void)	gauss_jackson/include/gauss_jackson_integrator_constructor.hh	144	1
jeod::GaussJacksonIntegratorConstructor::er7_utils::implements (er7_utils::Integration::ODEProblemType problem_type)	gauss_jackson/include/gauss_jackson_integrator_constructor.hh	150	1
jeod::GaussJacksonIntegratorConstructor::er7_utils::provides (er7_utils::Integration::ODEProblemType problem_type)	gauss_jackson/include/gauss_jackson_integrator_constructor.hh	161	1
jeod::GaussJacksonIntegratorConstructor::get_buffer_size (void)	gauss_jackson/include/gauss_jackson_integrator_constructor.hh	235	1
jeod::GaussJacksonIntegratorConstructor::get_transition_table_size (void)	gauss_jackson/include/gauss_jackson_integrator_constructor.hh	243	1
jeod::GaussJacksonOneState::GaussJacksonOneState ()	gauss_jackson/include/gauss_jackson_one_state.hh	84	1
jeod::GaussJacksonOneState::GaussJacksonOneState (double* first_in)	gauss_jackson/include/gauss_jackson_one_state.hh	92	1
jeod::GaussJacksonRationalCoefficients::GaussJacksonRationalCoefficients ()	gauss_jackson/include/gauss_jackson_rational_coeffs.hh	101	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJacksonSimpleSecondOrderODEIntegrator::er7_utils::GaussJacksonSimpleSecondOrderODEIntegrator (const er7_utils::IntegratorConstructor & priming_constructor, GaussJacksonIntegrationControls & controls, unsigned int size_in, er7_utils::IntegrationControls & priming_controls)	gauss_jackson/include/gauss_jackson_simple_second_order_ode_integrator.hh	103	1
jeod::GaussJacksonSimpleSecondOrderODEIntegrator::er7_utils::GaussJacksonSimpleSecondOrderODEIntegrator (const GaussJacksonSimpleSecondOrderODEIntegrator & src)	gauss_jackson/include/gauss_jackson_simple_second_order_ode_integrator.hh	126	1
jeod::GaussJacksonSimpleSecondOrderODEIntegrator::~GaussJacksonSimpleSecondOrderODEIntegrator ()	gauss_jackson/include/gauss_jackson_simple_second_order_ode_integrator.hh	139	1
jeod::GaussJacksonSimpleSecondOrderODEIntegrator::operator= (GaussJacksonSimpleSecondOrderODEIntegrator src)	gauss_jackson/include/gauss_jackson_simple_second_order_ode_integrator.hh	146	1
jeod::GaussJacksonSimpleSecondOrderODEIntegrator::SecondOrderODEIntegrator::swap (GaussJacksonSimpleSecondOrderODEIntegrator & other)	gauss_jackson/include/gauss_jackson_simple_second_order_ode_integrator.hh	158	1
jeod::GaussJacksonSimpleSecondOrderODEIntegrator::er7_utils::create_copy ()	gauss_jackson/include/gauss_jackson_simple_second_order_ode_integrator.hh	170	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJacksonSimpleSecondOrderODEIntegrator::reset_integrator ()	gauss_jackson/include/gauss_jackson_simple_second_order_ode_integrator.hh	180	1
jeod::GaussJacksonSimpleSecondOrderODEIntegrator::er7_utils::integrate (double dyn_dt, unsigned int target_stage, double const * ER7_UTILS_RESTRICT acc, double * ER7_UTILS_RESTRICT vel, double * ER7_UTILS_RESTRICT pos)	gauss_jackson/include/gauss_jackson_simple_second_order_ode_integrator.hh	189	1
jeod::GaussJacksonStateMachine::get_fsm_state ()	gauss_jackson/include/gauss_jackson_state_machine.hh	125	1
jeod::GaussJacksonStateMachine::get_max_history_size ()	gauss_jackson/include/gauss_jackson_state_machine.hh	131	1
jeod::GaussJacksonStateMachine::get_current_order ()	gauss_jackson/include/gauss_jackson_state_machine.hh	137	1
jeod::GaussJacksonStateMachine::get_history_length ()	gauss_jackson/include/gauss_jackson_state_machine.hh	143	1
jeod::GaussJacksonStateMachine::get_cycle_scale ()	gauss_jackson/include/gauss_jackson_state_machine.hh	149	1
jeod::GaussJacksonStateMachine::get_cycle_start_time ()	gauss_jackson/include/gauss_jackson_state_machine.hh	155	1
jeod::GaussJacksonStateMachine::get_at_downsample ()	gauss_jackson/include/gauss_jackson_state_machine.hh	161	1
jeod::GaussJacksonStateMachine::get_at_reinitialize ()	gauss_jackson/include/gauss_jackson_state_machine.hh	167	1
jeod::GaussJacksonStateMachine::get_at_order_change ()	gauss_jackson/include/gauss_jackson_state_machine.hh	173	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJacksonStateMachine::get_at_end_of_tour ()	gauss_jackson/include/gauss_jackson_state_machine.hh	179	1
jeod::GaussJacksonTwoState::GaussJacksonTwoState ()	gauss_jackson/include/gauss_jackson_two_state.hh	90	1
jeod::GaussJacksonTwoState::GaussJacksonTwoState (double* first_in, double* second_in)	gauss_jackson/include/gauss_jackson_two_state.hh	99	1
er7_utils::TwoDArray::TwoDArray ()	gauss_jackson/include/two_d_array.hh	87	1
er7_utils::TwoDArray::std::TwoDArray (const TwoDArray¡T¿& src)	gauss_jackson/include/two_d_array.hh	98	2
er7_utils::TwoDArray::~TwoDArray ()	gauss_jackson/include/two_d_array.hh	122	1
er7_utils::TwoDArray::operator= (TwoDArray¡T¿ src)	gauss_jackson/include/two_d_array.hh	131	1
er7_utils::TwoDArray::operator[] (int N)	gauss_jackson/include/two_d_array.hh	142	1
er7_utils::TwoDArray::operator[] (int N)	gauss_jackson/include/two_d_array.hh	152	1
er7_utils::TwoDArray::operator() ()	gauss_jackson/include/two_d_array.hh	163	1
er7_utils::TwoDArray::operator() ()	gauss_jackson/include/two_d_array.hh	174	1
er7_utils::TwoDArray::std::at (int N)	gauss_jackson/include/two_d_array.hh	205	3
er7_utils::TwoDArray::std::at (int N)	gauss_jackson/include/two_d_array.hh	219	3
er7_utils::TwoDArray::std::at (int N, int M)	gauss_jackson/include/two_d_array.hh	234	5
er7_utils::TwoDArray::std::at (int N, int M)	gauss_jackson/include/two_d_array.hh	250	5
er7_utils::TwoDArray::(std::allocate (std::size_t N, std::size_t M)	gauss_jackson/include/two_d_array.hh	267	5

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
er7_utils::TwoDArray::std:: rotate_down (int limit)	gauss_jackson/include/two_d_ array.hh	294	4
er7_utils::TwoDArray::std:: rotate_up (int limit)	gauss_jackson/include/two_d_ array.hh	313	4
er7_utils::TwoDArray::std:: downsample (int limit)	gauss_jackson/include/two_d_ array.hh	333	4
er7_utils::TwoDArray::std:: swap (TwoDArray¡T¿& other)	gauss_jackson/include/two_d_ array.hh	350	1
er7_utils::TwoDArray::swap (TwoDArray¡T¿& first, Two DArray¡T¿& second)	gauss_jackson/include/two_d_ array.hh	362	1
er7_utils::TwoDArray::er7_ utils::allocate_internal ()	gauss_jackson/include/two_d_ array.hh	402	1
er7_utils::TwoDArray::er7_ utils::deallocate_internal ()	gauss_jackson/include/two_d_ array.hh	411	1
jeod::GaussJackson CoefficientsPair::allocate_ arrays (int size)	gauss_jackson/src/gauss_ jackson_coefficients_pair.cc	36	1
jeod::GaussJackson CoefficientsPair::deallocate_ arrays ()	gauss_jackson/src/gauss_ jackson_coefficients_pair.cc	45	1
jeod::GaussJackson CoefficientsPair::swap (GaussJacksonCoefficients Pair & other)	gauss_jackson/src/gauss_ jackson_coefficients_pair.cc	53	1
jeod::GaussJackson CoefficientsPair::apply (int nelem, int ncoeff, double const * ER7_UTILS_REST RICT const * ER7_UTILS_ RESTRICT acc_hist, const GaussJacksonTwoState & state_sum)	gauss_jackson/src/gauss_ jackson_coefficients_pair.cc	62	4

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJackson CoefficientsPair::apply (int nelem, int ncoeff, double const * ER7_UTILS_REST RICT const * ER7_UTILS_ RESTRICT deriv_hist, const GaussJacksonOne State & state_sum)	gauss_jackson/src/gauss_ jackson_coefficients_pair.cc	95	4
jeod::GaussJackson CoefficientsPair::print (int order, std::ostream & stream)	gauss_jackson/src/gauss_ jackson_coefficients_pair.cc	122	3
jeod::GaussJacksonCoeffs::~ GaussJacksonCoeffs ()	gauss_jackson/src/gauss_ jackson_coeffs.cc	44	1
jeod::GaussJacksonCoeffs:: swap (GaussJacksonCoeffs & src)	gauss_jackson/src/gauss_ jackson_coeffs.cc	51	1
jeod::GaussJacksonCoeffs:: configure (unsigned int max_order_in)	gauss_jackson/src/gauss_ jackson_coeffs.cc	63	2
jeod::GaussJacksonCoeffs:: compute_coeffs (unsigned int order_in)	gauss_jackson/src/gauss_ jackson_coeffs.cc	83	2
jeod::std::operator¡¡ (std:: ostream& stream, const GaussJacksonCoeffs& coeff)	gauss_jackson/src/gauss_ jackson_coeffs.cc	142	2
jeod::GaussJacksonConfig:: default_configuration ()	gauss_jackson/src/gauss_ jackson_config.cc	39	1
jeod::GaussJacksonConfig:: standard_configuration ()	gauss_jackson/src/gauss_ jackson_config.cc	56	1
jeod::er7_utils::set_default_ config_values (const Gauss JacksonConfig & config)	gauss_jackson/src/gauss_ jackson_config.cc	73	17
jeod::er7_utils::validate_config (const GaussJacksonConfig & config)	gauss_jackson/src/gauss_ jackson_config.cc	160	11
jeod::GaussJacksonConfig:: validate_configuration (const GaussJacksonConfig & config)	gauss_jackson/src/gauss_ jackson_config.cc	233	3

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJackson GeneralizedDerivSecond OrderODEIntegrator:: GaussJacksonGeneralized DerivSecondOrderODE Integrator (const er7_utils:: IntegratorConstructor & primer_constructor, Gauss JacksonIntegrationControls & controls, unsigned int position_size, unsigned int velocity_size, const er7_utils::GeneralizedPosition DerivativeFunctions & deriv_funs, er7_utils:: IntegrationControls & priming_controls)	gauss_jackson/src/gauss_jackson_generalized_second_order_ode_integrator.cc	35	1
jeod::GaussJackson GeneralizedDerivSecond OrderODEIntegrator:: GaussJacksonGeneralized DerivSecondOrderODE Integrator (const Gauss JacksonGeneralizedDeriv SecondOrderODEIntegrator & src)	gauss_jackson/src/gauss_jackson_generalized_second_order_ode_integrator.cc	59	1
jeod::GaussJackson GeneralizedDerivSecond OrderODEIntegrator::~ GaussJacksonGeneralized DerivSecondOrderODE Integrator ()	gauss_jackson/src/gauss_jackson_generalized_second_order_ode_integrator.cc	79	1
jeod::GaussJackson GeneralizedDerivSecond OrderODEIntegrator::swap (GaussJacksonGeneralized DerivSecondOrderODE Integrator & other)	gauss_jackson/src/gauss_jackson_generalized_second_order_ode_integrator.cc	87	1
jeod::er7_utils::create_copy ()	gauss_jackson/src/gauss_jackson_generalized_second_order_ode_integrator.cc	101	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJackson IntegrationControls::Gauss JacksonIntegrationControls ()	gauss_jackson/src/gauss_jackson_integration_controls.cc	47	1
jeod::GaussJackson IntegrationControls::Gauss JacksonIntegrationControls (const er7_utils::Integrator Constructor & priming_constructor, const Gauss JacksonConfig & config_in)	gauss_jackson/src/gauss_jackson_integration_controls.cc	71	1
jeod::GaussJackson IntegrationControls::Gauss JacksonIntegrationControls (const GaussJackson IntegrationControls & src)	gauss_jackson/src/gauss_jackson_integration_controls.cc	101	2
jeod::GaussJackson IntegrationControls::~Gauss JacksonIntegrationControls (void)	gauss_jackson/src/gauss_jackson_integration_controls.cc	128	1
jeod::er7_utils::create_copy ()	gauss_jackson/src/gauss_jackson_integration_controls.cc	135	1
jeod::GaussJackson IntegrationControls::swap (GaussJacksonIntegration Controls & other)	gauss_jackson/src/gauss_jackson_integration_controls.cc	142	1
jeod::GaussJackson IntegrationControls::reset_integrator ()	gauss_jackson/src/gauss_jackson_integration_controls.cc	166	1
jeod::GaussJackson IntegrationControls:: integrate (double start_time, double sim_dt, er7_utils::TimeInterface & time_interface, er7_utils:: IntegratorInterface & integ_interface, er7_utils::Base IntegrationGroup & integ_group)	gauss_jackson/src/gauss_jackson_integration_controls.cc	178	14

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJackson IntegrationControls:: integrate_edit (er7_utils:: TimeInterface & time_ interface, er7_utils::Base IntegrationGroup & integ_ group)	gauss_jackson/src/gauss_ jackson_integration_ controls.cc	274	5
jeod::GaussJackson IntegrationControls:: integrate_gj (er7_utils::Time Interface & time_interface, er7_utils::BaseIntegration Group & integ_group)	gauss_jackson/src/gauss_ jackson_integration_ controls.cc	310	5
jeod::GaussJackson IntegrationControls::start_ cycle (double sim_dt)	gauss_jackson/src/gauss_ jackson_integration_ controls.cc	343	5
jeod::er7_utils::cast_to_gj_ controls (er7_utils:: IntegrationControls & controls)	gauss_jackson/src/gauss_ jackson_integrator_ constructor.cc	46	2
jeod::er7_utils::create_ constructor ()	gauss_jackson/src/gauss_ jackson_integrator_ constructor.cc	69	1
jeod::GaussJacksonIntegrator Constructor::GaussJackson IntegratorConstructor ()	gauss_jackson/src/gauss_ jackson_integrator_ constructor.cc	78	1
jeod::GaussJacksonIntegrator Constructor::GaussJackson IntegratorConstructor (const GaussJackson IntegratorConstructor & src)	gauss_jackson/src/gauss_ jackson_integrator_ constructor.cc	89	2
jeod::GaussJacksonIntegrator Constructor::~Gauss JacksonIntegrator Constructor ()	gauss_jackson/src/gauss_ jackson_integrator_ constructor.cc	104	1
jeod::GaussJacksonIntegrator Constructor::swap (Gauss JacksonIntegrator Constructor & src)	gauss_jackson/src/gauss_ jackson_integrator_ constructor.cc	111	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::GaussJacksonIntegratorConstructor::configure (const GaussJacksonConfig & config_in, er7_utils::Integration::Technique priming_technique)	gauss_jackson/src/gauss_jackson_integrator_constructor.cc	121	2
jeod::GaussJacksonIntegratorConstructor::configure (const GaussJacksonConfig & config_in, const er7_utils::IntegratorConstructor & priming_cotr_in)	gauss_jackson/src/gauss_jackson_integrator_constructor.cc	139	1
jeod::er7_utils::create_copy ()	gauss_jackson/src/gauss_jackson_integrator_constructor.cc	150	1
jeod::er7_utils::create_integration_controls ()	gauss_jackson/src/gauss_jackson_integrator_constructor.cc	159	2
jeod::er7_utils::create_first_order_ode_integrator (unsigned int size, er7_utils::IntegrationControls & controls)	gauss_jackson/src/gauss_jackson_integrator_constructor.cc	177	1
jeod::er7_utils::create_second_order_ode_integrator (unsigned int size, er7_utils::IntegrationControls & controls)	gauss_jackson/src/gauss_jackson_integrator_constructor.cc	198	1
jeod::er7_utils::create_generalized_deriv_second_order_ode_integrator (unsigned int position_size, unsigned int velocity_size, const er7_utils::GeneralizedPositionDerivativeFunctions & deriv_funs, er7_utils::IntegrationControls & controls)	gauss_jackson/src/gauss_jackson_integrator_constructor.cc	219	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::GaussJacksonRational Coefficients::configure_ adams_corrector (unsigned int nelem)	gauss_jackson/src/gauss_ jackson_rational_coeffs.cc	37	3
jeod::GaussJacksonRational Coefficients::construct_ stormer_cowell_corrector ()	gauss_jackson/src/gauss_ jackson_rational_coeffs.cc	60	3
jeod::GaussJacksonRational Coefficients::construct_ predictor ()	gauss_jackson/src/gauss_ jackson_rational_coeffs.cc	84	2
jeod::GaussJacksonRational Coefficients::convert_to_ ordinate_form (er7_utils::N ChooseM & n_choose_m, double * result)	gauss_jackson/src/gauss_ jackson_rational_coeffs.cc	108	4
jeod::GaussJacksonRational Coefficients::discard_extra_ terms (unsigned int nfront, unsigned int nback)	gauss_jackson/src/gauss_ jackson_rational_coeffs.cc	144	1
jeod::GaussJacksonRational Coefficients::displace_back ()	gauss_jackson/src/gauss_ jackson_rational_coeffs.cc	159	2
jeod::std::state_name (Fsm State state)	gauss_jackson/src/gauss_ jackson_state_machine.cc	33	6
jeod::GaussJacksonState Machine::GaussJackson StateMachine ()	gauss_jackson/src/gauss_ jackson_state_machine.cc	47	1
jeod::GaussJacksonState Machine::configure (const GaussJacksonConfig & config)	gauss_jackson/src/gauss_ jackson_state_machine.cc	79	2
jeod::GaussJacksonState Machine::reset ()	gauss_jackson/src/gauss_ jackson_state_machine.cc	97	1
jeod::GaussJacksonState Machine::set_bootstrap_ edit_redo_needed ()	gauss_jackson/src/gauss_ jackson_state_machine.cc	124	1
jeod::GaussJacksonState Machine::perform_step ()	gauss_jackson/src/gauss_ jackson_state_machine.cc	133	3

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::GaussJacksonStateMachine::transition_state ()	gauss_jackson/src/gauss_jackson_state_machine.cc	160	6
jeod::GaussJacksonStateMachine::exit_priming ()	gauss_jackson/src/gauss_jackson_state_machine.cc	205	2
jeod::GaussJacksonStateMachine::exit_bootstrap_edit ()	gauss_jackson/src/gauss_jackson_state_machine.cc	228	6
jeod::GaussJacksonStateMachine::exit_bootstrap_step ()	gauss_jackson/src/gauss_jackson_state_machine.cc	279	5
jeod::JeodIntegrationGroupOwner::~JeodIntegrationGroupOwner ()	include/jeod_integration_group.hh	100	1
jeod::JeodIntegrationGroup::need_first_step_derivatives (void)	include/jeod_integration_group.hh	166	1
jeod::JeodIntegrationGroup::update_from_owner (void)	include/jeod_integration_group.hh	176	1
jeod::JeodIntegrationGroup::er7_utils::merge_integrator_result (const er7_utils::IntegratorResult & new_result, er7_utils::IntegratorResult & merged_result)	include/jeod_integration_group.hh	184	1
jeod::JeodIntegrationGroup::respond_to_time_change ()	include/jeod_integration_group.hh	202	2
jeod::JeodIntegrationGroup::reset_body_integrators (void)	include/jeod_integration_group.hh	219	1
jeod::JeodIntegrationGroup::er7_utils::integrate_bodies (double cycle_dyndt, unsigned int target_stage)	include/jeod_integration_group.hh	232	1
jeod::JeodIntegrationGroup::T::reset_container (const T & container)	include/jeod_integration_group.hh	272	2

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::JeodIntegrationGroup:: er7_utils::integrate_ container (double dyn_dt, unsigned int target_stage, const T & container)	include/jeod_integration_ group.hh	290	2
jeod::Restartable2DSecond OrderIntegrator::~ Restartable2DSecondOrder Integrator ()	include/restartable_2d_ second_order_integrator.hh	102	1
jeod::Restartable2DSecond OrderIntegrator::er7_utils:: create_integrator (const er7_ utils::IntegratorConstructor & generator, er7_utils:: IntegrationControls & controls)	include/restartable_2d_ second_order_integrator.hh	111	1
jeod::Restartable2DSecond OrderIntegrator::destroy_ integrator ()	include/restartable_2d_ second_order_integrator.hh	123	1
jeod::Restartable2DSecond OrderIntegrator::er7_utils:: integrate (double dyn_dt, unsigned int target_stage, double const * ER7_UTILS_ RESTRICT accel, double * ER7_UTILS_RESTRICT velocity, double * ER7_UTI LS_RESTRICT position)	include/restartable_2d_ second_order_integrator.hh	132	1
jeod::Restartable2DSecond OrderIntegrator::reset_ integrator ()	include/restartable_2d_ second_order_integrator.hh	160	1
jeod::Restartable2DSecond OrderIntegrator::simple_ restore ()	include/restartable_2d_ second_order_integrator.hh	171	1
jeod::RestartableScalarFirst OrderODEIntegrator::~ RestartableScalarFirst OrderODEIntegrator ()	include/restartable_state_ integrator.hh	107	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::RestartableScalarFirstOrderODEIntegrator::er7_utils::create_integrator (const er7_utils::IntegratorConstructor & generator, er7_utils::IntegrationControls & controls)	include/restartable_state_integrator.hh	112	1
jeod::RestartableScalarFirstOrderODEIntegrator::destroy_integrator ()	include/restartable_state_integrator.hh	124	1
jeod::RestartableScalarFirstOrderODEIntegrator::er7_utils::integrate (double dyn_dt, unsigned int target_stage, double * ER7_UTILS_RESTRICT xdot, double * ER7_UTILS_RESTRICT x)	include/restartable_state_integrator.hh	132	1
jeod::RestartableScalarFirstOrderODEIntegrator::reset_integrator ()	include/restartable_state_integrator.hh	157	1
jeod::RestartableScalarFirstOrderODEIntegrator::simple_restore ()	include/restartable_state_integrator.hh	168	1
jeod::RestartableT3SecondOrderODEIntegrator::~RestartableT3SecondOrderODEIntegrator ()	include/restartable_state_integrator.hh	229	1
jeod::RestartableT3SecondOrderODEIntegrator::er7_utils::create_integrator (const er7_utils::IntegratorConstructor & generator, er7_utils::IntegrationControls & controls)	include/restartable_state_integrator.hh	234	1
jeod::RestartableT3SecondOrderODEIntegrator::destroy_integrator ()	include/restartable_state_integrator.hh	246	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::RestartableT3SecondOrderODEIntegrator::er7_utils::integrate (double dyn_dt, unsigned int target_stage, double const * ER7_UTILS_RESTRICT accel, double * ER7_UTILS_RESTRICT velocity, double * ER7_UTILS_RESTRICT position)	include/restartable_state_integrator.hh	254	1
jeod::RestartableT3SecondOrderODEIntegrator::reset_integrator ()	include/restartable_state_integrator.hh	282	1
jeod::RestartableT3SecondOrderODEIntegrator::simple_restore ()	include/restartable_state_integrator.hh	293	1
jeod::RestartableSO3SecondOrderODEIntegrator::~RestartableSO3SecondOrderODEIntegrator ()	include/restartable_state_integrator.hh	357	1
jeod::RestartableSO3SecondOrderODEIntegrator::GeneralizedSecondOrderODETechnique::create_integrator (GeneralizedSecondOrderODETechnique::TechniqueType technique_in, const er7_utils::IntegratorConstructor & generator, er7_utils::IntegrationControls & controls)	include/restartable_state_integrator.hh	365	3
jeod::RestartableSO3SecondOrderODEIntegrator::GeneralizedSecondOrderODETechnique::destroy_integrator ()	include/restartable_state_integrator.hh	399	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::RestartableSO3SecondOrderODEIntegrator::er7_utils::integrate (double dyn_dt, unsigned int target_stage, double const * ER7_UTILS_RESTRICT accel, double * ER7_UTILS_RESTRICT velocity, double * ER7_UTILS_RESTRICT position)	include/restartable_state_integrator.hh	410	1
jeod::RestartableSO3SecondOrderODEIntegrator::reset_integrator ()	include/restartable_state_integrator.hh	438	1
jeod::RestartableSO3SecondOrderODEIntegrator::GeneralizedSecondOrderODETechnique::simple_restore ()	include/restartable_state_integrator.hh	449	3
jeod::RestartableStateIntegrator::~RestartableStateIntegrator ()	include/restartable_state_integrator_templates.hh	179	1
jeod::RestartableStateIntegrator::er7_utils::create_integrator (const er7_utils::IntegratorConstructor & generator, er7_utils::IntegrationControls & controls)	include/restartable_state_integrator_templates.hh	187	2
jeod::RestartableStateIntegrator::er7_utils::destroy_integrator ()	include/restartable_state_integrator_templates.hh	211	2
jeod::RestartableStateIntegrator::clear_integrator_reference ()	include/restartable_state_integrator_templates.hh	221	1
jeod::RestartableStateIntegrator::set_integrator_reference (IntegratorType *& integ_ptr)	include/restartable_state_integrator_templates.hh	231	1
jeod::RestartableStateIntegrator::simple_restore ()	include/restartable_state_integrator_templates.hh	243	2

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::RestartableStateIntegrator::RestartableStateIntegrator ()	include/restartable_state_integrator_templates.hh	260	1
jeod::RestartableStateIntegrator::RestartableStateIntegrator (IntegratorType *& integ_ref)	include/restartable_state_integrator_templates.hh	269	1
jeod::RestartableStateIntegrator::simple_restore_internal (IntegratorType * integrator_ptr JEOD_UNUSED)	include/restartable_state_integrator_templates.hh	296	1
jeod::RestartableFirstOrderODEIntegrator::RestartableStateIntegrator¡er7_utils::RestartableFirstOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	339	1
jeod::RestartableFirstOrderODEIntegrator::er7_utils::RestartableFirstOrderODEIntegrator (er7_utils::FirstOrderODEIntegrator *& integ_ref)	include/restartable_state_integrator_templates.hh	346	1
jeod::RestartableFirstOrderODEIntegrator::~RestartableFirstOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	356	1
jeod::RestartableFirstOrderODEIntegrator::er7_utils::create_integrator_internal (const er7_utils::IntegratorConstructor & generator, er7_utils::IntegrationControls & controls)	include/restartable_state_integrator_templates.hh	367	1
jeod::RestartableSecondOrderODEIntegrator::~RestartableSecondOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	408	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::RestartableSecondOrderODEIntegrator::RestartableStateIntegrator¡er7_utils::RestartableSecondOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	417	1
jeod::RestartableSecondOrderODEIntegrator::er7_utils::RestartableSecondOrderODEIntegrator (er7_utils::SecondOrderODEIntegrator *& integ_ref)	include/restartable_state_integrator_templates.hh	424	1
jeod::RestartableSimpleSecondOrderODEIntegrator::RestartableSimpleSecondOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	466	1
jeod::RestartableSimpleSecondOrderODEIntegrator::er7_utils::RestartableSimpleSecondOrderODEIntegrator (er7_utils::SecondOrderODEIntegrator *& integ_ref)	include/restartable_state_integrator_templates.hh	473	1
jeod::RestartableSimpleSecondOrderODEIntegrator::~RestartableSimpleSecondOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	483	1
jeod::RestartableSimpleSecondOrderODEIntegrator::er7_utils::create_integrator_internal (const er7_utils::IntegratorConstructor & generator, er7_utils::IntegrationControls & controls)	include/restartable_state_integrator_templates.hh	494	1
jeod::RestartableGeneralizedDerivSecondOrderODEIntegrator::RestartableGeneralizedDerivSecondOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	542	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::RestartableGeneralizedDerivSecondOrderODEIntegrator::er7_utils::RestartableGeneralizedDerivSecondOrderODEIntegrator (er7_utils::SecondOrderODEIntegrator *& integ_ref)	include/restartable_state_integrator_templates.hh	549	1
jeod::RestartableGeneralizedDerivSecondOrderODEIntegrator::~RestartableGeneralizedDerivSecondOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	559	1
jeod::RestartableGeneralizedDerivSecondOrderODEIntegrator::er7_utils::create_integrator_internal (const er7_utils::IntegratorConstructor & generator, er7_utils::IntegrationControls & controls)	include/restartable_state_integrator_templates.hh	570	1
jeod::RestartableGeneralizedDerivSecondOrderODEIntegrator::er7_utils::simple_restore_internal (er7_utils::SecondOrderODEIntegrator * integrator_ptr)	include/restartable_state_integrator_templates.hh	583	2
jeod::RestartableGeneralizedStepSecondOrderODEIntegrator::RestartableGeneralizedStepSecondOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	634	1
jeod::RestartableGeneralizedStepSecondOrderODEIntegrator::er7_utils::RestartableGeneralizedStepSecondOrderODEIntegrator (er7_utils::SecondOrderODEIntegrator *& integ_ref)	include/restartable_state_integrator_templates.hh	641	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
jeod::RestartableGeneralizedStepSecondOrderODEIntegrator::~RestartableGeneralizedStepSecondOrderODEIntegrator ()	include/restartable_state_integrator_templates.hh	651	1
jeod::RestartableGeneralizedStepSecondOrderODEIntegrator::er7_utils::create_integrator_internal (const er7_utils::IntegratorConstructor & generator, er7_utils::IntegrationControls & controls)	include/restartable_state_integrator_templates.hh	662	1
jeod::RestartableGeneralizedStepSecondOrderODEIntegrator::er7_utils::simple_restore_internal (er7_utils::SecondOrderODEIntegrator * integrator_ptr)	include/restartable_state_integrator_templates.hh	675	2
jeod::LsodeControlDataInterface::~LsodeControlDataInterface (void)	lsode/include/lsode_control_data_interface.hh	134	1
jeod::LsodeControlDataInterface::is_corrector_method_functional_iteration ()	lsode/include/lsode_control_data_interface.hh	151	1
jeod::LsodeFirstOrderODEIntegrator::get_re_entry_point ()	lsode/include/lsode_first_order_ode_integrator.hh	194	1
jeod::LsodeIntegrationControls::~LsodeIntegrationControls ()	lsode/include/lsode_integration_controls.hh	104	1
jeod::LsodeIntegratorConstructor::get_class_name (void)	lsode/include/lsode_integrator_constructor.hh	123	1
jeod::LsodeIntegratorConstructor::(er7_utils::implements (er7_utils::Integration::ODEProblemType problem_type)	lsode/include/lsode_integrator_constructor.hh	129	2

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::LsodeIntegrator Constructor::(er7_utils:: provides (er7_utils:: Integration::ODEProblem Type problem_type)	lsode/include/lsode_ integrator_constructor.hh	139	2
jeod::LsodeIntegrator Constructor::get_transition_ table_size (void)	lsode/include/lsode_ integrator_constructor.hh	204	1
jeod::LsodeSecondOrderODE Integrator::get_re_entry_ point ()	lsode/include/lsode_second_ order_ode_integrator.hh	109	1
jeod::LsodeSecondOrderODE Integrator::reset_integrator ()	lsode/include/lsode_second_ order_ode_integrator.hh	139	1
LsodeControlDataInterface:: LsodeControlDataInterface ()	lsode/src/lsode_control_data_ interface.cc	51	1
LsodeControlDataInterface:: LsodeControlDataInterface (const LsodeControlData Interface & src)	lsode/src/lsode_control_data_ interface.cc	80	1
LsodeControlDataInterface:: check_interface_data ()	lsode/src/lsode_control_data_ interface.cc	107	27
LsodeControlDataInterface:: allocate_arrays ()	lsode/src/lsode_control_data_ interface.cc	285	5
LsodeControlDataInterface:: destroy_allocated_arrays ()	lsode/src/lsode_control_data_ interface.cc	330	2
LsodeControlDataInterface:: set_rel_tol (int index, double value)	lsode/src/lsode_control_data_ interface.cc	346	7
LsodeControlDataInterface:: set_abs_tol (int index, double value)	lsode/src/lsode_control_data_ interface.cc	397	7
LsodeDataJacobianPrep:: LsodeDataJacobianPrep ()	lsode/src/lsode_data_ classes.cc	54	1
LsodeDataStode::LsodeData Stode ()	lsode/src/lsode_data_ classes.cc	69	1
LsodeDataArrays::LsodeData Arrays ()	lsode/src/lsode_data_ classes.cc	86	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
LsodeDataArrays::allocate_arrays (unsigned int num_odes_in, LsodeControlDataInterface::CorrectorMethod corrector_method)	lsode/src/lsode_data_classes.cc	102	7
LsodeDataArrays::destroy_allocated_arrays ()	lsode/src/lsode_data_classes.cc	197	4
LsodeFirstOrderODEIntegrator::integrator_core ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	54	10
LsodeFirstOrderODEIntegrator::integrator_reset_method_coeffs ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	233	5
LsodeFirstOrderODEIntegrator::integrator_test_stepsize_change ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	273	2
LsodeFirstOrderODEIntegrator::integrator_reset_yh ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	297	4
LsodeFirstOrderODEIntegrator::integrator_predict ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	333	6
LsodeFirstOrderODEIntegrator::integrator_reset_iteration_loop_part1 ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	374	2
LsodeFirstOrderODEIntegrator::integrator_reset_iteration_loop_part2 ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	392	2
LsodeFirstOrderODEIntegrator::integrator_corrector_iteration ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	412	12
LsodeFirstOrderODEIntegrator::integrator_corrector_failed_part1 ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	509	3
LsodeFirstOrderODEIntegrator::integrator_corrector_failed_part2 ()	lsode/src/lsode_first_order_ode_integrator__integrator.cc	539	6

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
LsodeFirstOrderODE Integrator::integrator_ corrector_converged ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	588	10
LsodeFirstOrderODE Integrator::integrator_error_ test_failed ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	659	6
LsodeFirstOrderODE Integrator::integrator_ compute_new_order_prep ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	701	3
LsodeFirstOrderODE Integrator::integrator_ compute_new_order ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	745	9
LsodeFirstOrderODE Integrator::integrator_ compute_new_order_check_ step_error ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	815	4
LsodeFirstOrderODE Integrator::integrator_set_ new_order ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	838	2
LsodeFirstOrderODE Integrator::integrator_fail_ reset_order_1_part1 ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	862	3
LsodeFirstOrderODE Integrator::integrator_fail_ reset_order_1_part2 ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	906	3
LsodeFirstOrderODE Integrator::integrator_ wrapup ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	931	2
LsodeFirstOrderODE Integrator::integrator_ terminate ()	lsode/src/lsode_first_order_ ode_integrator__ integrator.cc	949	1
er7_utils::integrate (double dyn_dt, unsigned int target_ stage JEOD_UNUSED, double const * y_dot_in, double * y_in)	lsode/src/lsode_first_order_ ode_integrator__manager.cc	57	16
LsodeFirstOrderODE Integrator::process_entry_ point_cycle_start ()	lsode/src/lsode_first_order_ ode_integrator__manager.cc	255	6

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
LsodeFirstOrderODE Integrator::manager_ initialize_calculation_part1 ()	lsode/src/lsode_first_order_ ode_integrator__manager.cc	328	2
LsodeFirstOrderODE Integrator::manager_ initialize_calculation_part2 ()	lsode/src/lsode_first_order_ ode_integrator__manager.cc	381	18
LsodeFirstOrderODE Integrator::manager_check_ stop_conditions ()	lsode/src/lsode_first_order_ ode_integrator__manager.cc	514	7
LsodeFirstOrderODE Integrator::manager_ integration_loop_part1 ()	lsode/src/lsode_first_order_ ode_integrator__manager.cc	635	4
LsodeFirstOrderODE Integrator::manager_ integration_loop_part2 ()	lsode/src/lsode_first_order_ ode_integrator__manager.cc	686	6
LsodeFirstOrderODE Integrator::manager_ integration_loop_part3 ()	lsode/src/lsode_first_order_ ode_integrator__manager.cc	744	9
LsodeFirstOrderODE Integrator::reset_integrator ()	lsode/src/lsode_first_order_ ode_integrator__manager.cc	864	4
LsodeFirstOrderODE Integrator::manager_set_ calculation_phase_eq_2_ reload ()	lsode/src/lsode_first_order_ ode_integrator__manager.cc	890	1
LsodeFirstOrderODE Integrator::calculate_epsilon ()	lsode/src/lsode_first_order_ ode_integrator__support.cc	54	2
LsodeFirstOrderODE Integrator::calculate_ integration_coefficients ()	lsode/src/lsode_first_order_ ode_integrator__support.cc	72	12
LsodeFirstOrderODE Integrator::interpolate_y ()	lsode/src/lsode_first_order_ ode_integrator__support.cc	241	7
LsodeFirstOrderODE Integrator::jacobian_prep_ init ()	lsode/src/lsode_first_order_ ode_integrator__support.cc	321	11

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
LsodeFirstOrderODE Integrator::jacobian_prep_ loop ()	lsode/src/lsode_first_order_ ode_integrator__support.cc	507	9
LsodeFirstOrderODE Integrator::jacobian_prep_ wrap_up ()	lsode/src/lsode_first_order_ ode_integrator__support.cc	613	15
LsodeFirstOrderODE Integrator::linear_chord_ iteration ()	lsode/src/lsode_first_order_ ode_integrator__support.cc	722	11
LsodeFirstOrderODE Integrator::load_ew_values ()	lsode/src/lsode_first_order_ ode_integrator__support.cc	794	9
LsodeFirstOrderODE Integrator::LsodeFirstOrder ODEIntegrator (void)	lsode/src/lsode_first_order_ ode_integrator__utility.cc	52	5
LsodeFirstOrderODE Integrator::LsodeFirstOrder ODEIntegrator (const LsodeControlDataInterface & data_in, er7_utils:: IntegrationControls & controls, unsigned int size)	lsode/src/lsode_first_order_ ode_integrator__utility.cc	121	5
LsodeFirstOrderODE Integrator::~LsodeFirst OrderODEIntegrator ()	lsode/src/lsode_first_order_ ode_integrator__utility.cc	197	2
LsodeFirstOrderODE Integrator::update_control_ data ()	lsode/src/lsode_first_order_ ode_integrator__utility.cc	209	4
LsodeFirstOrderODE Integrator::create_copy ()	lsode/src/lsode_first_order_ ode_integrator__utility.cc	244	1
LsodeFirstOrderODE Integrator::magnitude_of_ weighted_array (const double * v)	lsode/src/lsode_first_order_ ode_integrator__utility.cc	260	2
LsodeFirstOrderODE Integrator::magnitude_of_ weighted_array (unsigned int index, double ** v)	lsode/src/lsode_first_order_ ode_integrator__utility.cc	285	2

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
LsodeFirstOrderODE Integrator::gauss_elim_ factor ()	lsode/src/lsode_first_order_ ode_integrator__utility.cc	305	10
LsodeFirstOrderODE Integrator::linear_solver ()	lsode/src/lsode_first_order_ ode_integrator__utility.cc	384	7
LsodeFirstOrderODE Integrator::index_of_max_ magnitude (unsigned int num_points, double ** array, int start_ix)	lsode/src/lsode_first_order_ ode_integrator__utility.cc	453	3
LsodeFirstOrderODE Integrator::load_derivatives (double * derivs)	lsode/src/lsode_first_order_ ode_integrator__utility.cc	492	2
LsodeGeneralizedDerivSecond OrderODEIntegrator::Lsode GeneralizedDerivSecond OrderODEIntegrator (void)	lsode/src/lsode_generalized_ second_order_ode_ integrator.cc	51	1
LsodeGeneralizedDerivSecond OrderODEIntegrator::Lsode GeneralizedDerivSecond OrderODEIntegrator (const LsodeControlDataInterface & data_in, er7_utils:: IntegrationControls & controls, const er7_utils:: GeneralizedPosition DerivativeFunctions & deriv_funs, unsigned int position_size, unsigned int velocity_size)	lsode/src/lsode_generalized_ second_order_ode_ integrator.cc	60	1
LsodeGeneralizedDerivSecond OrderODEIntegrator::Lsode GeneralizedDerivSecond OrderODEIntegrator (const LsodeGeneralizedDeriv SecondOrderODEIntegrator & src JEOD_UNUSED)	lsode/src/lsode_generalized_ second_order_ode_ integrator.cc	79	1
LsodeGeneralizedDerivSecond OrderODEIntegrator:: create_copy ()	lsode/src/lsode_generalized_ second_order_ode_ integrator.cc	94	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
LsodeGeneralizedDerivSecondOrderODEIntegrator::~LsodeGeneralizedDerivSecondOrderODEIntegrator (void)	lsode/src/lsode_generalized_second_order_ode_integrator.cc	110	1
er7_utils::integrate (double dyn_dt, unsigned int, double const * ER7_UTILS_RESTRICT accel, double * ER7_UTILS_RESTRICT velocity, double * ER7_UTILS_RESTRICT position)	lsode/src/lsode_generalized_second_order_ode_integrator.cc	119	5
LsodeIntegrationControls::LsodeIntegrationControls (void)	lsode/src/lsode_integration_controls.cc	49	1
LsodeIntegrationControls::LsodeIntegrationControls (unsigned int num_stages JEOD_UNUSED)	lsode/src/lsode_integration_controls.cc	55	1
LsodeIntegrationControls::create_copy ()	lsode/src/lsode_integration_controls.cc	60	1
LsodeIntegrationControls::integrate (double starttime, double sim_dt, er7_utils::TimeInterface & time_interface, er7_utils::IntegratorInterface & integ_interface, er7_utils::BaseIntegrationGroup & integ_group)	lsode/src/lsode_integration_controls.cc	73	6
LsodeIntegratorConstructor::LsodeIntegratorConstructor (const LsodeIntegratorConstructor & src)	lsode/src/lsode_integrator_constructor.cc	54	1
er7_utils::create_constructor (void)	lsode/src/lsode_integrator_constructor.cc	65	1
er7_utils::create_copy (void)	lsode/src/lsode_integrator_constructor.cc	73	1
er7_utils::create_integration_controls (void)	lsode/src/lsode_integrator_constructor.cc	83	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
er7_utils::create_first_order_ode_integrator (unsigned int size, er7_utils::IntegrationControls & controls)	lsode/src/lsode_integrator_constructor.cc	93	1
er7_utils::create_second_order_ode_integrator (unsigned int size, er7_utils::IntegrationControls & controls)	lsode/src/lsode_integrator_constructor.cc	112	1
er7_utils::create_generalized_deriv_second_order_ode_integrator (unsigned int position_size, unsigned int velocity_size, const er7_utils::GeneralizedPositionDerivativeFunctions & deriv_funs, er7_utils::IntegrationControls & controls)	lsode/src/lsode_integrator_constructor.cc	130	1
LsodeSecondOrderODEIntegrator::LsodeSecondOrderODEIntegrator (void)	lsode/src/lsode_second_order_ode_integrator.cc	51	1
LsodeSecondOrderODEIntegrator::LsodeSecondOrderODEIntegrator (const LsodeControlDataInterface & data_in, er7_utils::IntegrationControls & controls, unsigned int size)	lsode/src/lsode_second_order_ode_integrator.cc	66	1
LsodeSecondOrderODEIntegrator::LsodeSecondOrderODEIntegrator (const LsodeControlDataInterface & data_in, er7_utils::IntegrationControls & controls, const er7_utils::GeneralizedPositionDerivativeFunctions & deriv_funs, unsigned int position_size, unsigned int velocity_size)	lsode/src/lsode_second_order_ode_integrator.cc	87	1

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	File	Line	ECC
LsodeSecondOrderODE Integrator::~LsodeSecond OrderODEIntegrator (void)	lsode/src/lsode_second_order_ ode_integrator.cc	117	2
LsodeSimpleSecondOrderOD EIntegrator::LsodeSimple SecondOrderODEIntegrator (void)	lsode/src/lsode_simple_ second_order_ode_ integrator.cc	49	1
LsodeSimpleSecondOrderOD EIntegrator::LsodeSimple SecondOrderODEIntegrator (const LsodeControlData Interface & data_in, er7_ utils::IntegrationControls & controls, unsigned int size)	lsode/src/lsode_simple_ second_order_ode_ integrator.cc	56	1
LsodeSimpleSecondOrderOD EIntegrator::create_copy ()	lsode/src/lsode_simple_ second_order_ode_ integrator.cc	67	1
er7_utils::integrate (double dyn_dt, unsigned int target_ stage JEOD_UNUSED, double const * ER7_UTILS_ RESTRICT accel, double * ER7_UTILS_RESTRICT velocity, double * ER7_UTI LS_RESTRICT position)	lsode/src/lsode_simple_ second_order_ode_ integrator.cc	80	3
jeod::GeneralizedSecondOrder ODETechnique::is_ provided_by (const er7_ utils::IntegratorConstructor & generator, Technique Type technique)	src/generalized_second_order_ ode_technique.cc	44	4
jeod::GeneralizedSecondOrder ODETechnique::validate_ technique (const er7_utils:: IntegratorConstructor & generator, TechniqueType technique, const char * file, unsigned int line, const char * requester, const char * name)	src/generalized_second_order_ ode_technique.cc	70	10

Continued on next page

Table 5.3: Cyclomatic Complexity (continued)

Method	**File**	**Line**	**ECC**
jeod::JeodIntegrationGroup:: register_classes ()	src/jeod_integration_group.cc	51	1
jeod::JeodIntegrationGroup:: JeodIntegrationGroup ()	src/jeod_integration_group.cc	62	1
jeod::JeodIntegrationGroup:: JeodIntegrationGroup (JeodIntegrationGroup Owner & owner, er7_utils:: IntegratorConstructor & integ_cotr, JeodIntegrator Interface & integ_inter, JeodIntegrationTime & time_mngr)	src/jeod_integration_group.cc	78	1
jeod::JeodIntegrationGroup::~ JeodIntegrationGroup ()	src/jeod_integration_group.cc	101	2
jeod::JeodIntegrationGroup:: add_integrable_object (er7_ utils::IntegrableObject & integrable_object)	src/jeod_integration_group.cc	112	4
jeod::JeodIntegrationGroup:: remove_integrable_object (er7_utils::IntegrableObject & integrable_object)	src/jeod_integration_group.cc	143	3
jeod::JeodIntegrationGroup:: initialize_group ()	src/jeod_integration_group.cc	172	2
jeod::JeodIntegrationTime:: JeodIntegrationTime ()	src/jeod_integration_time.cc	43	1
jeod::JeodIntegrationTime::~ JeodIntegrationTime ()	src/jeod_integration_time.cc	58	1
jeod::JeodIntegrationTime:: add_time_change_subscriber (TimeChangeSubscriber & subscriber)	src/jeod_integration_time.cc	69	2
jeod::JeodIntegrationTime:: remove_time_change_ subscriber (TimeChange Subscriber & subscriber)	src/jeod_integration_time.cc	93	2
jeod::JeodIntegrationTime:: notify_time_change_ subscribers ()	src/jeod_integration_time.cc	118	2

Appendix A

Integration Techniques

A.1 Overview

The appendices that follow each describe a single technique for solving an an initial value problem. Some techniques natively address a first order ODE, others, a second order ODE. A first order ODE solver can readily be adapted to solve a second order ODE, but at the expense of ignoring some of the geometry of the problem. The converse is not true for a second order ODE solver. Most techniques, at their heart, address a scalar problem. These scalar-based techniques can readily be adapted to solving a vector problem, but once again at the expense of ignoring some of the geometry of the problem. A small number (only one in JEOD 3.0) explicitly address a vector ODE problem and do take advantage of the connections between the components of the vector.

A.2 Classification

A numerical integration technique can be classified in a number of ways:

- Whether the technique is a solver for a first or second order ODE.
- Whether the technique takes advantage of the vectorial nature of the problem.
- Whether the technique uses derivatives from previous integration intervals. The single step integrators do not use prior history. The multistep integrations do, which means these techniques need to be "primed" by some other integration technique.
- Whether the technique adaptively splits the integration interval into smaller cycles. LSODE is the only adaptive technique in JEOD 3.0.
- Whether the technique subdivides an integration cycle into stages. The Runge Kutta intergrators are multistage. The linear multistep integrators are single stage (but typically are multistep).

A.3 Implemented and Provided Techniques

The implementation of an integration technique includes classes for solving initial value problems and an integrator constructor that creates instances of those solvers. The integrator constructor derives from the the ER7 Utilities Integration model base class `er7_utils::IntegratorConstructor`. This base class defines four overridable methods for constructing state integrators:

`create_first_order_ode_integrator` Creates an integrator for a first order ODE initial value problem. These integrators take a state vector and its time derivative as arguments. The goal of the integration is to advance the state vector to some desired end time.

`create_second_order_ode_integrator` Creates an integrator for a second order ODE initial value problem. These integrators take a state vector and its first and second derivative as arguments. The goal of the integration is to advance the state vector and its first derivative to some desired end time.

`create_generalized_deriv_second_order_ode_integrator` Creates an integrator for a generalized derivative second order ODE initial value problem. These integrators take a generalized position vector, a generalized velocity vector, and the time derivative of generalized velocity as arguments. The time derivative of generalized position is computed by some function of generalized position and velocity; this function is provided as an argument to the constructor function. The goal of the integration is to advance generalized position and velocity to some desired end time.

`create_generalized_step_second_order_ode_integrator` Creates an integrator for a second order Lie group initial value problem.

The base implementation of each of these is to generate an error message and return a null pointer. Using a null pointer to perform integration will cause the program to crash. The "Implemented and Provided Techniques" section of each of the technique-specific appendices that follow contains a table that indicates which of these four interfaces the technique supports. This support can be

- Direct – The integrator constructor returns a solver implemented in the context of the technique,
- Indirect – The integrator constructor returns a solver that is compatible with the technique, or
- Non-existent – The technique does not override the default implementation.

A.4 Mathematical Description

The "Mathematical Description" section of each of the technique-specific appendices that follow contains a description of the mathematics behind the technique. References are provided if possible.

A.4.1 First Order ODE

Most of the provided techniques target a scalar first order ODE. Adapting such a technique to a vectorial first order ODE is trivial: Simply apply the technique individually to each element of the vector. This comes at a cost of ignoring some of the geometry of the problem. LSODE specifically targets a vector first order ODE and can take geometry into account.

Some of the provided techniques target second order ODEs. These techniques typically cannot be used to solve first order ODE initial value problems. These techniques provide a surrogate first order solver that is compatible with the second order ODE technique when asked to create a solver for a first order ODE initial value problem.

A.4.2 Second Order ODE

All of the provided techniques provide solvers for second order ODEs. A first order ODE solver can readily be adapted to solving a second order ODE initial value problem, either by state doubling or by applying the technique to position first and velocity second. The remaining techniques explicitly solve a second order ODE initial value problem.

A.4.3 Generalized Position / Generalized Velocity ODE

All of the provided techniques provide solvers for generalized position / generalized velocity initial value problems. The constructor for the solver takes the functions that computes the first and second derivatives of generalized position as arguments. The same mechanisms used to adapt a first order ODE solver to a second order ODE problem can be employed using these functions. These functions can similarly be used by second order ODE solvers to adapt that solver to a generalized position / generalized velocity problem.

A.4.4 Lie Group Second Order ODE

Lie group solvers only exist for some of the integration techniques. A Lie group problem can always be recast as a generalized position / generalized velocity problem, so techniques that do not provide a Lie group integrator can still be used to solve these kinds of problems (but at the cost of accuracy).

A.5 Usage Instructions

The "Usage Instructions" section of each of the technique-specific appendices that follow contains a description of how to use the technique.

Appendix B

Euler's Method

B.1 Overview

Euler's method is the simplest of the integrators. The technique addresses initial value problems for first order ODEs. Euler's method propagates state to the end of an integration interval in one step by assuming that the values of the derivatives over the integration interval are equal to the values at the start of the interval.

B.2 Classification

Euler's method is a single step, single stage, single cycle integrator for scalar-valued first order ODEs. The technique exhibits a global truncation error that is proportional to the step size. As is the case for other solvers for scalar-valued first order ODEs, Euler's method can easily be adapted to solving vector-valued initial value problems of arbitrary order.

B.3 Implemented and Provided Techniques

Table B.1 lists the types of initial value problems that can be solved using Euler's method. As noted in the table, Euler's method implementations exist for all four types of initial value problems addressed by the ER7 Utilities integration suite.

Table B.1: Euler Method Initial Value Problems

Problem Type	**Support**
`FirstOrderODE`	Implemented
`SimpleSecondOrderODE`	Implemented
`GeneralizedDerivSecondOrderODE`	Implemented
`GeneralizedStepSecondOrderODE`	Implemented

B.4 Mathematical Description

Euler's method is the simplest of techniques for numerically solving a scalar first order ODE initial value problem given by equation (B.1).

$$\begin{aligned} \frac{ds(t)}{dt} &= f(t, s(t)) \\ s(t_0) &= s_0 \end{aligned} \tag{B.1}$$

Euler's method propagates state from time t to $t + \Delta t$ by assuming that the derivative is constant between t and $t + \Delta t$ and is equal to the value at the start of the integration interval [3]:

$$s(t + \Delta t) = s(t) + f(t, s(t))\Delta t \tag{B.2}$$

Note that in numerical integration literature, it is the integration technique that calls the derivative function $f(t, s(t))$. That is not the case with Trick integration, and hence with the ER7 utilities integrators. The derivatives are instead inputs to the integration function.

B.4.1 First Order ODE

The ER7 Utilities implementation of Euler's method applies equation (B.2) to each element of a vector-valued first order ODE initial value problem.

B.4.2 Second Order ODE

The ER7 Utilities implementation of Euler's method applies equation (B.2) to each element of the position vector and then to each element of the velocity vector to solve a vector-valued second order ODE initial value problem.

B.4.3 Generalized Position / Generalized Velocity ODE

The ER7 Utilities implementation of Euler's method applies equation (B.2) to each element of the position vector (using the generalized position derivative function to compute the derivative of the position vector) and then to each element of the velocity vector to solve a vector-valued generalized position / generalized velocity initial value problem.

B.4.4 Lie Group Second Order ODE

Applying Euler's method to a second order Lie group problem involves advancing generalized position using the Lie group exponential map step function applied against the initial generalized position and generalized velocity. Generalized velocity is once again advanced per the basic Euler method.

The exponential map step function `expmap_step`$(\Delta\boldsymbol{\theta}, \boldsymbol{x})$ advances position via $\text{expmap}(\Delta\boldsymbol{\theta}) \cdot \boldsymbol{x}$. In this expression, the exponential is the exponential map function that maps from the tangent Lie algebra space at x to the Lie group. The multiplication of this quantity with the input generalized position is performed per the group operator of the Lie group.

$$\begin{aligned} \Delta\boldsymbol{\theta} &= \boldsymbol{v}(t)\Delta t \\ \boldsymbol{x}(t+\Delta t) &= \texttt{expmap\_step}(\Delta\boldsymbol{\theta}, \boldsymbol{x}(t)) \\ \boldsymbol{v}(t+\Delta t) &= \boldsymbol{v}(t) + \dot{\boldsymbol{v}}(t)\Delta t \end{aligned} \tag{B.3}$$

B.5 Implementation

TBS

B.6 Usage Instructions

To use Euler's method to integrate the dynamics of a monolithic JEOD simulation (*i.e.*, one in which the dynamics manager performs the integration), set the `jeod_integ_opt` element of the simulation's `DynManagerInit` object to `trick.Integration.Euler` in the appropriate Python input file.

The method is provided for backwards compatibility reasons and because Euler's method forms the basis of almost all numerical integration techniques.

It is recommended that Euler's method not be used in any simulation. Euler's method is highly inaccurate and rather unstable. The only advantage of this technique is that it needs just one evaluation of the derivative function per integration cycle. More accurate techniques that similarly require just one derivative evaluation per integration cycle exist. Use one of these alternatives in lieu of using Euler's method, or use a higher order technique. Any other technique is better than Euler's method.

Appendix C

Symplectic Euler's Method

C.1 Overview

The symplectic Euler method (also known as the semi-implicit Euler method, the semi-explicit Euler method, the Euler-Cromer method) is the simplest of integrators for second order ODE initial value problems. The method propagates position and velocity by assuming a constant acceleration and constant velocity across the integration interval. The technique advances velocity to the end of the integration interval using the acceleration at the start of the interval and advances position using the updated velocity.

C.2 Classification

The symplectic Euler method is a single step, single stage, single cycle integrator for scalar-valued second order ODEs. The only derivatives required are those at the start of an integration cycle, which makes this method a single step integrator. The algorithm proceeds immediately to the end of the cycle, which makes this method a single stage integrator. The algorithm does not subdivide an integration tour into multiple cycles, which makes this method a single cycle integrator.

As is the case for other solvers for scalar-valued second order ODEs, the symplectic Euler method can easily be adapted to solving vector-valued second order ODE initial value problems. In cannot be applied to initial value problems of arbitrary order.

Like the basic Euler method, the symplectic Euler exhibits a local truncation error that is proportional to the square of the step size and thus exhibits a global truncation error that is proportional to the step size. The symplectic Euler method is thus a first order integrator in terms of accuracy.

C.3 Implemented and Provided Techniques

Table C.1 lists the types of initial value problems that can be solved using the symplectic Euler method. As noted in the table, the symplectic Euler method provides solvers for all four types of initial value problems addressed by the ER7 Utilities integration suite.

Table C.1: Symplectic Euler Method Initial Value Problems

Problem Type	**Support**
`FirstOrderODE`	Provided
`SimpleSecondOrderODE`	Implemented
`GeneralizedDerivSecondOrderODE`	Implemented
`GeneralizedStepSecondOrderODE`	Implemented

C.4 Mathematical Description

The symplectic Euler method propagates velocity from time t to $t + \Delta t$ by assuming a constant acceleration between t and $t + \Delta t$ that is equal to the value at the start of the integration interval. The method then propagates positon by assuming a constant velocity over the integration interval that is equal to the value at the end of the interval:

$$\begin{aligned} v(t+\Delta t) &= v(t) + \Delta t f(t, x(t), v(t)) \\ x(t+\Delta t) &= x(t) + \Delta t v(t+\Delta t) \end{aligned} \tag{C.1}$$

C.5 First Order ODE

The symplectic Euler method explicitly addresses second order ODEs, meaning that it cannot be used to solve an initial value problem for a first order ODE. The basic Euler method is used as a surrogate for symplectic Euler.

C.6 Second Order ODE

The ER7 Utilities implementation of the symplectic Euler method applies equation (C.1) to each element of the position and velocity vectors.

C.7 Generalized Position / Generalized Velocity ODE

TBS

C.8 Lie Group Second Order ODE

TBS

C.9 Usage Instructions

To use the symplectic Euler method to integrate the dynamics of a monolithic JEOD simulation (*i.e.*, one in which the dynamics manager performs the integration), set the `jeod_integ_opt` element of the simulation's `DynManagerInit` object to `trick.Integration.SymplecticEuler` in the appropriate Python input file.

Appendix D

Beeman's Algorithm

Beeman's algorithm is a two-stage predictor-corrector method. It requires knowledge of the velocity-derivative from one integration cycle period before the start of this integration cycle. It is, therefore, also a multi-step method, and requires a primer to launch it. In JEOD, we use the RK2 integrator as the primer.

Beeman's algorithm explicitly distinguishes generalized position and generalized velocity. The equations that govern propagation via Beeman's Algorithm are given by equations (D.1) to (D.3)[1].

Priming (one simulation step):

$$\dot{\boldsymbol{v}}_{-1} = \dot{\boldsymbol{v}}(t_i - \Delta t, x(t_i - \Delta t), \boldsymbol{v}(t_i - \Delta t)) \tag{D.1}$$

Stage 0 (predictor):

$$\begin{aligned}
t_1 &= t_f = t_i + \Delta t \\
\dot{\boldsymbol{v}}_0 &= \dot{\boldsymbol{v}}(t_i, x(t_i), \boldsymbol{v}(t_i)) \\
\bar{\boldsymbol{v}}_1 &= \boldsymbol{v}(t_i) + \Delta t \left(\frac{2}{3}\dot{\boldsymbol{v}}_0 - \frac{1}{6}\dot{\boldsymbol{v}}_{-1} \right) \\
\bar{\dot{\boldsymbol{x}}}_1 &= \dot{\boldsymbol{x}}(\boldsymbol{x}(t_i), \bar{\boldsymbol{v}}_1) \\
\bar{\dot{\boldsymbol{v}}}_1 &= \frac{3}{2}\dot{\boldsymbol{v}}_0 - \frac{1}{2}\dot{\boldsymbol{v}}_{-1} \\
\boldsymbol{x}(t_1) &= \boldsymbol{x}(t_i) + \Delta t\, \bar{\dot{\boldsymbol{x}}}_1 \\
\boldsymbol{v}(t_1) &= \boldsymbol{v}(t_i) + \Delta t\, \bar{\dot{\boldsymbol{v}}}_1
\end{aligned} \tag{D.2}$$

Stage 1 (corrector):

$$\begin{aligned}
t_2 &= t_1 = t_f = t_i + \Delta t \\
\dot{\boldsymbol{v}}_1 &= \dot{\boldsymbol{v}}(t_1, x(t_1), \boldsymbol{v}(t_1)) \\
\bar{\dot{\boldsymbol{v}}}_2 &= \frac{1}{3}\dot{\boldsymbol{v}}_1 + \frac{5}{6}\dot{\boldsymbol{v}}_0 - \frac{1}{6}\dot{\boldsymbol{v}}_{-1} \\
\boldsymbol{x}(t_f) &= \boldsymbol{x}(t_1) \\
\boldsymbol{v}(t_f) &= \boldsymbol{v}(t_i) + \Delta t\, \bar{\dot{\boldsymbol{v}}}_2 \\
\dot{\boldsymbol{v}}_{-1} &= \dot{\boldsymbol{v}}_0
\end{aligned} \tag{D.3}$$

Beeman's Algorithm offers significant improvements in accuracy over the symplectic Euler method. Beeman's algorithm is a second order method while the Euler methods are first order. Like the fourth-order Adams-Bashforth-Moulton method, Beeman's Algorithm is not self-starting. It relies on the derivatives from the previous time step. JEOD uses the RK2 method as a primer for Beeman's Algorithm.

Appendix E

Second Order Nyström-Lear

Appendix F

Position Verlet

Appendix G

Heun's Method

The RK2 method is the simplest of the predictor-corrector methods. The prediction of the final state is made using Euler's method, then the derivatives computed for the final state are combined with the derivatives of the initial state to produce a mean over the interval. This mean value is then used to correct the final state.

This is a second-order, two-stage, single-step, single-cycle algorithm.

The first stage involves an Euler step to the end of the interval. The second stage computes the mean value as the average of the state derivative at the start of the interval and the state derivative at the end of the interval (as computed with the state from the first step). Table G.1 specifies the Butcher tableau for Heun's method.

Table G.1: Heun's Method Butcher Tableau

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array}$$

Equations (G.1) and (G.2) describe the equations that govern propagation using Heun's method.

Stage 0 (predictor):

$$\begin{aligned} t_1 &= t_f = t_i + \Delta t \\ \dot{\boldsymbol{s}}_0 &= \dot{\boldsymbol{s}}(t_i, \boldsymbol{s}(t_i)) \\ \boldsymbol{s}_1(t_1) &= \boldsymbol{s}(t_i) + \Delta t\, \dot{\boldsymbol{s}}_0 \end{aligned} \tag{G.1}$$

Stage 1 (corrector):

$$\begin{aligned} t_2 &= t_f = t_1 = t_i + \Delta t \\ \dot{\boldsymbol{s}}_1 &= \dot{\boldsymbol{s}}(t_1, \boldsymbol{s}(t_1)) \\ \bar{\dot{\boldsymbol{s}}} &= \frac{\dot{\boldsymbol{s}}_0 + \dot{\boldsymbol{s}}_1}{2} \\ \boldsymbol{s}(t_f) &= \boldsymbol{s}(t) + \Delta t\, \bar{\dot{\boldsymbol{s}}} \end{aligned} \tag{G.2}$$

There are an infinite number of Runge Kutta methods for any given order and number of stages. For example, another second order two stage Runge Kutta method is the midpoint method. The midpoint method takes an Euler step to the midpoint of the interval and uses the derivative at this midpoint as the mean value.

Appendix H

Midpoint Method

Appendix I

Velocity Verlet

Appendix J

Modified Midpoint

Appendix K

Fourth Order Adams-Bashforth-Moulton

Adams-Bashforth-Moulton methods combine an explicit Adams-Bashforth integrator as a predictor and an implicit Adams-Moulton integrator as a corrector. The fourth order Adams-Bashforth-Moulton method implemented in JEOD uses the four step Adams-Bashforth integrator as a predictor and the three step Adams-Moulton integrator as a corrector.

K.0.0.1 Maintaining the Derivative History

All but the simplest of Adams-Bashforth-Moulton methods require a history of prior derivatives. JEOD uses the fourth order Adams-Bashforth-Moulton method which requires four sets of prior derivatives.

On the first step of an integration cycle the JEOD implementation of the ABM4 method saves the input derivatives in a derivative history buffer. How derivatives are saved depends on whether the requisite set of four prior derivatives have been gathered. The method operates in priming mode until the requisite set of four prior derivatives have been gathered.

In priming mode, the derivatives are stored consecutively in the slot designated by a primer counter which advances from zero to three as derivatives are gathered. Once primed, the derivative history buffer is treated as a circular array. New derivatives are copied into the slot occupied by the oldest data and pointers are advanced circularly.

K.0.0.2 Integrating State

The state is advanced on each integration step. The primer is used to advance the state while the method is in priming mode. Once the requisite four sets of derivatives from the primer have been gathered, the ABM4 integrator uses a two stage predictor-corrector technique to advance the state. The equations that govern propagation via the ABM4 method are given by equations (K.1) and (K.2)[8].

Stage 0 (predictor):

$$\begin{aligned}
\bar{\dot{\boldsymbol{s}}}_{AB} &= \frac{55\dot{\boldsymbol{s}}_0 - 59\dot{\boldsymbol{s}}_{-1} + 37\dot{\boldsymbol{s}}_{-2} - 9\dot{\boldsymbol{s}}_{-3}}{24} \\
t_1 &= t_f = t_i + \Delta t \\
\boldsymbol{s}_1(t_1) &= \boldsymbol{s}(t) + \Delta t\, \bar{\dot{\boldsymbol{s}}}_{AB}
\end{aligned} \tag{K.1}$$

Stage 1 (corrector):

$$\begin{aligned}
\dot{\boldsymbol{s}}_1 &= \dot{\boldsymbol{s}}(t_1, \boldsymbol{s}(t_1)) \\
\bar{\dot{\boldsymbol{s}}}_{AM} &= \frac{9\dot{\boldsymbol{s}}_1 + 19\dot{\boldsymbol{s}}_0 - 5\dot{\boldsymbol{s}}_{-1} + \dot{\boldsymbol{s}}_{-2}}{24} \\
t_2 &= t_1 = t_f = t_i + \Delta t \\
\boldsymbol{s}(t_2) &= \boldsymbol{s}(t) + \Delta t\, \bar{\dot{\boldsymbol{s}}}_{AM}
\end{aligned} \tag{K.2}$$

K.0.0.3 ABM4 Primer

The simplest of the Adams-Bashforth-Moulton methods combines explicit and implicit Euler integration. This simplest implementation does not rely on historical data. All other Adams-Bashforth-Moulton methods, including the fourth order method implemented in JEOD, do rely on historical data; they are not self-starting. Some other technique must be used to prime the method. Once primed the Adams-Bashforth-Moulton methods are self-sustaining. As both the fourth order Adams-Bashforth-Moulton method and the fourth order Runge Kutta method have fourth order accuracy, JEOD uses the RK4 method as a primer for the ABM4 method.

Appendix L

Classical Fourth Order Runge Kutta (RK4)

L.1 Overview

The classical fourth order Runge Kutta integrator is one of the most widely used techniques for solving initial value problems. The technique addresses initial value problems for first order ODEs. As the name suggests, this technique is a Runge Kutta method, a single step (i.e., no past history), multi-stage integrator.

L.2 Classification

TBS

L.3 Implemented and Provided Techniques

Table L.1 lists the types of initial value problems that can be used in conjunction with an RK4-based integrator constructor. As noted in the table, RK4 implementations exist for all four types of initial value problems addressed by the ER7 Utilities integration suite.

Table L.1: RK4 Initial Value Problems

Problem Type	**Support**
`FirstOrderODE`	Implemented
`SimpleSecondOrderODE`	Implemented
`GeneralizedDerivSecondOrderODE`	Implemented
`GeneralizedStepSecondOrderODE`	Implemented

L.4 Mathematical Description

The ER7 utilities RungeKutta4 integration technique is the standard fourth order, four stage Runge Kutta method. Table L.2 specifies the Butcher tableau for this method.

Table L.2: RK4 Butcher Tableau

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

The equations that govern propagation via the standard RK4 method are given by equations (L.1) to (L.4).

Stage 0:

$$\begin{aligned} t_1 &= t_i + \frac{\Delta t}{2} \\ \ldots_0 &= \dot{s}(t_i, s(t_i)) \\ s_1(t_1) &= s(t_i) + \frac{\Delta t}{2} \ldots_0 \end{aligned} \tag{L.1}$$

Stage 1:

$$\begin{aligned} t_2 = t_1 &= t_i + \frac{\Delta t}{2} \\ \ldots_1 &= \dot{s}(t_1, s_1(t_1)) \\ s_2(t_2) &= s(t_i) + \frac{\Delta t}{2} \ldots_1 \end{aligned} \tag{L.2}$$

Stage 2:

$$\begin{aligned} t_3 = t_f &= t_i + \Delta t \\ \ldots_2 &= \dot{s}(t_2, s_1(t_2)) \\ s_3(t_3) &= s(t_i) + \Delta t \ldots_2 \end{aligned} \tag{L.3}$$

Stage 3 (final):

$$\begin{aligned} \ldots_3 &= \dot{s}(t_3, s_1(t_3)) \\ t_4 = t_3 = t_f &= t_i + \Delta t \\ \bar{\ldots} &= \frac{\ldots_0 + 2\dot{s}_1 + 2\ldots_2 + \dot{s}_3}{6} \\ s(t_f) &= s(t_i) + \Delta t\, \bar{\ldots} \end{aligned} \tag{L.4}$$

L.5 First Order ODE

The ER7 Utilities implementation of the fourth order Runge-Kutta technique applies equations (L.1) to (L.4) to each element of a vector-valued first order ODE initial value problem.

L.6 Second Order ODE

The ER7 Utilities implementation of the fourth order Runge-Kutta technique applies equations (L.1) to (L.4) to each element of the position vector and then to each element of the velocity vector to solve a vector-valued second order ODE initial value problem.

L.7 Generalized Position / Generalized Velocity ODE

The ER7 Utilities implementation of the fourth order Runge-Kutta technique applies equations (L.1) to (L.4) to each element of the position vector (using the generalized position derivative function to compute the derivative of the position vector) and then to each element of the velocity vector to solve a vector-valued generalized position / generalized velocity initial value problem.

L.8 Lie Group Second Order ODE

TBS

L.9 Usage Instructions

The classical fourth order Runge-Kutta technique is the default integration technique for a monolithic JEOD simulation.

To specifically select this technique, set the `jeod_integ_opt` element of the simulation's `DynManagerInit` object to `trick.Integration.RungeKutta4` in the appropriate Python input file.

Appendix M

Fourth Order Runge-Kutta-Gill

Appendix N

Runge-Kutta-Fehlberg 4/5

Appendix O

Runge-Kutta-Fehlberg 7/8

Appendix P

Gauss-Jackson

P.1 Derivation of Gauss-Jackson Algorithm

Here, we present the derivation of the coefficients used in this Gauss-Jackson implementation. The derivation follows that of Berry and Healy [2], generalized to provide user-specified order; the method is further generalized to provide the bootstrapping off the RK4 primer.

Although in JEOD we are dealing typically with 3-vectors, the 3 elements in those vectors are mutually independent and integration must be carried out term-by-term. In this analysis, we will be dealing with the integration of single-dimension values, e.g. only the x-component of some vector.

P.1.1 Nomenclature

We will be representing the state value as x_n: the value of variable x at time-step n. For our purposes, we are assuming a constant time-step, h.

We are looking for a method by which the zeroth derivative (x_n) and the first derivative ($\dot{x}_n$) can be written in terms of previously known values $\{x_i, \dot{x}_i, \ddot{x}_i\}$ for $i < n$.

The current state, and its derivative, will be expressed as a combination of those previous states, each multiplied by some predetermined coefficients. The derivation of the coefficients for the Gauss-Jackson integration scheme requires a systematic derivation of Adams-Moulton corrector coefficients, Adams-Bashforth predictor coefficients, Störmer-Cowell corrector coefficients, and Störmer predictor coefficients.

The Summed-Adams method is used for the generation of the first-derivative state (e.g. velocity).

The Gauss-Jackson method is used for the generation of the zeroth-derivative state (e.g. position).

1. The raw coefficients:

   - c_i code:am Adams-Moulton coefficient (used in derivation of A_{Ci} and A_{Ii})
   - c'_i code:ab Adams-Bashforth coefficients (used in derivation of A_{Pi})

- q_i code:*sc* Störmer-Cowell coefficients (used in derivation of G_{Ci} and G_{Ii})
- q'_i code:*sp* Störmer coefficients (used in derivation of G_{Pi})

2. Intermediate coefficients, identified in the derivation but not used in the code:

   - g_i Gauss-Jackson corrector coefficients
   - g'_i Gauss-Jackson predictor coefficients
   - σ_i Summed-Adams corrector coefficients
   - σ'_i Summed-Adams predictor coefficients

3. The processed coefficients in ordinate form:

   - A_{Ci} code:*sa_corrector_coeff* Summed-Adams corrector coefficients (equation P.30)
   - A_{Ii} code:*sa_corrector_coeff* Summed-Adams initializing coefficients (equation P.40)
   - A_{Pi} code:*sa_predictor_coeff* Summed-Adams predictor coefficients (equation P.33)
   - G_{Ci} code:*gj_corrector_coeff* Gauss-Jackson corrector coefficients (equation P.35)
   - G_{Ii} code:*gj_corrector_coeff* Gauss-Jackson initializing coefficients (equation P.46)
   - G_{Pi} code:*gj_predictor_coeff* Gauss-Jackson predictor coefficients (equation P.37)

4. The operators:

   - The backward-difference operator, ∇:

$$\nabla x_n = x_n - x_{n-1} \tag{P.1}$$

   - The displacement operator, E:

$$Ex_n = x_{n+1} \tag{P.2}$$

   - The differentiation operator, D:

$$Dx_n = \dot{x}_n \tag{P.3}$$

   - The corrector integration operator, L:

$$x_n = x_{n-1} + hL(\dot{x}_n) \tag{P.4}$$

   - The predictor integration operator, J:

$$x_n = x_{n-1} + hJ(\dot{x}_{n-1}) \tag{P.5}$$

P.1.2 Applications of the Operators

- Consider multiple applications of the backward-difference operator:

$$\nabla^2 x_n = \nabla x_n - \nabla x_{n-1} = (x_n - x_{n-1}) - (x_{n-1} - x_{n-2}) = x_n - 2x_{n-1} + x_{n-2}$$
$$\nabla^3 x_n = \nabla^2 x_n - \nabla^2 x_{n-1} = (x_n - 2x_{n-1} + x_{n-2}) - (x_{n-1} - 2x_{n-2} + x_{n-3})$$

In general, this follows the binomial:

$$\nabla^i x_n = \sum_{j=0}^{j=i} (-1)^j \binom{i}{j} x_{n-j} \tag{P.6}$$

where

$$\binom{i}{j} = \frac{i!}{j!(i-j)!}$$

- Consider repetitive application of the displacement operator, E.

  $$E^m x_n = x_{n+m}$$

  This can be expanded as a Taylor series:

  $$E^m x_n = x_n + mh\dot{x}_n + \frac{(mh)^2}{2!}\ddot{x}_n + \frac{(mh)^3}{3!}\dddot{x}_n + ...$$

- Consider the relation between E and D. Substituting the differentiation operator, D, the Taylor expansion can be written as

  $$E^m x_n \;=\; \left(1 + mhD + \frac{(mh)^2}{2!}D^2 + \frac{(mh)^3}{3!}D^3 + ...\right) x_n \;=\; e^{mhD} x_n \tag{P.7}$$

  Consequently, E and D are related by

  $$\ln E = hD \tag{P.8}$$

- Consider the relation between E and ∇.

  $$E^{-1} x_n \;=\; x_{n-1} \;=\; x_n - \nabla x_n$$

  Consequently,

  $$E^{-1} = 1 - \nabla$$

  $$E = \frac{1}{1-\nabla} \tag{P.9}$$

- Relation between D and ∇. Thus,

  $$hD = \ln E = -\ln(1-\nabla) \tag{P.10}$$

  and

  $$\frac{D^{-1}}{h} = -\frac{1}{\ln(1-\nabla)} \tag{P.11}$$

We need expressions for the integration operators, J and L, in terms of ∇, since this is the operator that allows us to access previous states.

P.1.3 Derivation of the Coefficients

P.1.3.1 Adams-Moulton Coefficients

Consider the corrector operator, L, first; it is the easier of the two.

The defining equation for the corrector operator (equation P.4):

$$x_n = x_{n-1} + hL(\dot{x}_n)$$

Application of the backward-difference operator gives $\nabla x_n = x_n - x_{n-1}$, which can be re-written as $\nabla x_n = hL(\dot{x}_n)$, with the derivative expressable in terms of the differentiation operator, D.

$$\nabla x_n = hL(D(x_n))$$

Consequently, from equation P.11 we have a relation between L, D, and ∇:

$$L = \frac{\nabla D^{-1}}{h} = -\frac{\nabla}{\ln(1-\nabla)} \tag{P.12}$$

Recognizing that the Taylor expansion of $\ln(1-x) = -\left(x + \frac{x^2}{2} + \frac{x^3}{3} + ...\right)$, we can write:

$$L = -\frac{1}{1 + \frac{\nabla}{2} + \frac{\nabla^2}{3} + ...} \tag{P.13}$$

We are looking for an expression for L in terms of ∇. Writing L as a power expansion of ∇:

$$L = \sum_{i=0}^{\infty} c_i \nabla^i \tag{P.14}$$

and substituting into equation P.13 and rearranging, we have:

$$\left(\sum_{i=0}^{\infty} c_i \nabla^i\right)\left(1 + \frac{\nabla}{2} + \frac{\nabla^2}{3} + ...\right) = 1$$

Collecting powers of ∇ yields:

$c_0 = 1$

$\frac{c_0}{2} + c_1 = 0$

$\frac{c_0}{3} + \frac{c_1}{2} + c_2 = 0$

etc.

In general,

$$c_n = -\sum_{i=0}^{n-1} \frac{c_i}{n+1-i} \tag{P.15}$$

These represent the Adams-Moulton corrector coefficients, and are used in generating the final set of coefficients for the corrector side of the summed-Adams integrator, which is used in this implementation of the Gauss-Jackson integrator for generating the state of the first derivative (e.g. velocity).

These coefficients can be used to correct the current state, following calculation of the derivative of the state based on the previous calculation of the current state, for example:

$$x_n = x_{n-1} + h\left(\sum_{i=0}^{\infty} c_i \nabla^i\right)(\dot{x}_n) \tag{P.16}$$

The selected order of the integrator determines how many of these coefficients are needed; for reasons which may become apparent later, this implementation requires *N+2* of these coefficients for an integrator of order N.

P.1.3.2 Adams-Bashforth Coefficients

Next consider the predictor integration operator, J.

The defining equation for the predictor operator (equation P.5):

$$x_{n+1} = x_n + h\ J(\dot{x}_n)$$

Application of the backward-difference operator gives $\nabla x_{n+1} = x_{n+1} - x_n$, which can be re-written as $\nabla x_{n+1} = h\ J(\dot{x}_n)$, with the derivative expressable in terms of the differentiation operator, D, and the displacement operator, E.

$$\nabla x_{n+1} = h\ J(D(x_n)) = h\ J(D(E^{-1}(x_{n+1})))$$

Rearranging and substituting from equations P.11 and P.9 gives:

$$J = \frac{\nabla E D^{-1}}{h} = -\frac{\nabla}{(1-\nabla)\ln(1-\nabla)} \tag{P.17}$$

Recognizing the similarities between equations P.17 and P.12, the predictor integration operator can be written in terms of the backward-difference operator and the corrector integrator operator (for which we already have an expression in terms of the backward-difference operator):

$$J = (1-\nabla)^{-1} L \tag{P.18}$$

J can now be written as a power expansion in ∇, with coefficients evaluated by writing L, and $(1-\nabla)^{-1}$ as power expansions (see equation P.14 for expansion of L):

$$J = \left(\sum_{i=0}^{\infty} c'_i \nabla^i\right) = \left(1 + \nabla + \nabla^2 + \nabla^3 + ...\right)\left(\sum_{i=0}^{\infty} c_i \nabla^i\right)$$

Hence, by comparing powers of ∇, the coefficients for J can be related to those for L:

$$c'_i \;=\; \sum_{j=0}^{i} c_j \;=\; c'_{i-1} + c_i$$

These are the Adams-Bashforth predictor coefficients, and are used in generating the final set of coefficients for the predictor side of the summed-Adams integrator, which is used in this implementation of the Gauss-Jackson integrator for generating the state of the first derivative (e.g. velocity).

These coefficients can be used to predict the next state, following calculation of the derivative of the final, converged, value of the current state. For example:

$$x_{n+1} = x_n + h\left(\sum_{i=0}^{\infty} c'_i \nabla^i\right)(\dot{x}_n) \tag{P.19}$$

P.1.3.3 Störmer-Cowell Coefficients

To obtain the coefficients for the second-order integrators, we need to apply the appropriate integrator operators twice.

First, consider the expansion of L^2, with the expression for L from equation P.14:

$$L^2 = \sum_{i=0}^{\infty} q_i \nabla^i = (c_0 + c_1\nabla + c_2\nabla^2 + ...)(c_0 + c_1\nabla + c_2\nabla^2 + ...) \tag{P.20}$$

It is apparent that

$$q_i = \sum_{j=0}^{i} c_j c_{i-j} \tag{P.21}$$

The double-application of the corrector integrator operator, L yields:

$$L^2(\ddot{x}_n) \;=\; \frac{x_n - 2x_{n-1} + x_{n-2}}{h^2}$$

Consequently,

$$x_n = 2x_{n-1} - x_{n-2} + h^2\left(\sum_{i=0}^{\infty} q_i \nabla^i\right)\ddot{x}_n \tag{P.22}$$

This provides a convenient formula for obtaining the double integral; it is considered a *corrector* because it relies on the second derivative at some point in order to generate the zeroth derivative at the same point.

The values q_i are the Störmer-Cowell coefficients, and are used in generating the final set of coefficients for the corrector side of the Gauss-Jackson integrator, which is used in this implementation of the Gauss-Jackson integrator for generating the state of the zeroth derivative (e.g. position).

P.1.3.4 Störmer Coefficients

The comparative predictor coefficients can be obtained by application of the displacement operator again, just as we generated the Adams-Bashforth coefficients from the Adams-Moulton coefficients.

$$L^2(E(\ddot{x}_n)) = \frac{x_{n+1} - 2x_n + x_{n-1}}{h^2}$$

As with the derivation of the Adams-Bashforth coefficients, this can be expressed as a power series in ∇:

$$L^2(E(\ddot{x}_n)) = \left(\sum_{i=0}^{\infty} q_i' \nabla^i\right)(\ddot{x}_n) \Rightarrow$$

$$L^2 E = L^2(1-\nabla)^{-1} = \left(\sum_{i=0}^{\infty} q_i \nabla^i\right)(1 + \nabla + \nabla^2 + \ldots) = \left(\sum_{i=0}^{\infty} q_i' \nabla^i\right)$$

with

$$q_i' \;=\; \sum_{j=0}^{i} q_j \;=\; q_{i-1}' + q_i \tag{P.23}$$

$$q_0' = q_0 = 1$$

These coefficients are used to predict the next state, and for generating the predictor coefficients for the Gauss-Jackson integrator.

$$x_{n+1} = 2x_n - x_{n-1} + h^2(\sum_{i=0}^{\infty} q_i' \nabla^i)\,(\ddot{x}_n) \tag{P.24}$$

This provides a convenient formula for obtaining the double integral; it is considered a *predictor* because it relies on the second derivative at some point in order to generate the zeroth derivative at a point in the future.

The values q_i' are the Störmer coefficients, and are used in generating the final set of coefficients for the predictor side of the Gauss-Jackson integrator, which is used in this implementation of the Gauss-Jackson integrator for generating the state of the zeroth derivative (e.g. position).

Note the similarity between this predictor form (equation P.24) and the corrector form (equation P.22) shifted by one:

$$x_{n+1} = 2x_n - x_{n-1} + h^2(\sum_{i=0}^{\infty} q_i \nabla^i)\ddot{x}_{n+1}$$

This allows the easy recognition that:

$$\sum_{i=0}^{\infty} q_i \nabla^i \ddot{x}_{n+1} = \sum_{i=0}^{\infty} q'_i \nabla^i \ddot{x}_n \tag{P.25}$$

P.1.4 Summed Forms of the Integrators

The summed forms express the desired target value in terms of the appropriate derivatives and the *summation operator*, ∇^{-1}

P.1.4.1 Summed-Adams Corrector

The summed-Adams corrector is derived directly from the Adams-Moulton expressions (equation P.16).

$$x_n = x_{n-1} + h\left(\sum_{i=0}^{\infty} c_i \nabla^i\right)(\dot{x}_n)$$

The two zeroth derivative terms can be collected on the left hand side, and written in terms of the backward-difference operator, ∇.

$$\nabla(x_n) = h\left(\sum_{i=0}^{\infty} c_i \nabla^i\right)(\dot{x}_n)$$

Now applying ∇^{-1} to both sides, and pulling the first term ($c_0 = 1$) out of the summation, yields:

$$x_n = h\left(\nabla^{-1} + \sum_{i=0}^{\infty} c_{i+1} \nabla^i\right)(\dot{x}_n)$$

The coefficients in the sum for the summed-Adams corrector are just those for the Adams-Moulton corrector, only offset by one position:

$$x_n = h\left(\nabla^{-1} + \sum_{i=0}^{\infty} \sigma_i \nabla^i\right)(\dot{x}_n) \tag{P.26}$$

with $\sigma_i = c_{i+1}$

P.1.4.2 Summed-Adams Predictor

The summed-Adams predictor is derived from the Adams-Bashforth expressions in the same way as the corrector is derived from the Adams-Moulton expressions.

$$x_{n+1} = h\left(\nabla^{-1} + \sum_{i=0}^{\infty} \sigma_i' \nabla^i\right)(\dot{x}_n) \tag{P.27}$$

with $\sigma_i' = c_{i+1}'$

P.1.4.3 Gauss-Jackson Corrector

The derivation of the Gauss-Jackson double integration expressions comes from the expressions of Störmer and Cowell in much the same way that the summed Adams were just derived.

Notice the symmetry between L^2 and ∇^2:

$$\nabla x_n = hL(\dot{x}_n)$$

$$\nabla^2 x_n = h^2 L^2(\ddot{x}_n)$$

Thus, from equation P.20:

$$\nabla^2(x_n) = h^2\left(\sum_{i=0}^{\infty} q_i \nabla^i\right)(\ddot{x}_n)$$

and x_n can be written as

$$x_n = h^2\nabla^{-2}\left(\sum_{i=0}^{\infty} q_i \nabla^i\right)(\ddot{x}_n) = h^2\left(\sum_{i=0}^{\infty} q_i \nabla^{i-2}\right)(\ddot{x}_n)$$

Consider the first two terms in the summation, as defined in the derivation of the Störmer-Cowell coefficients.

$$\begin{aligned} q_0 &= c_0 c_0 = 1 \\ q_1 &= c_0 c_1 + c_1 c_0 = -1 \end{aligned}$$

Pulling these out of the summation term yields:

$$x_n = h^2\left(\nabla^{-2} - \nabla^{-1} + \sum_{i=0}^{\infty} q_{i+2}\nabla^i\right)(\ddot{x}_n) = h^2\left(\nabla^{-2}(1-\nabla) + \sum_{i=0}^{\infty} q_{i+2}\nabla^i\right)(\ddot{x}_n)$$

Recall that $E^{-1} = (1 - \nabla)$ (equation P.9)

$$x_n \;=\; h^2 \left(\nabla^{-2} E^{-1} + \sum_{i=0}^{\infty} q_{i+2} \nabla^i \right) (\ddot{x}_n)$$

Thus

$$x_n = h^2 \left(\nabla^{-2}(\ddot{x}_{n-1}) + \sum_{i=0}^{\infty} g_i \nabla^i (\ddot{x}_n) \right) \tag{P.28}$$

with $g_i = q_{i+2}$.

This is the Gauss-Jackson corrector formulation, with coefficients equal to those from the Störmer-Cowell, but displaced by two. The additional term at the front has to be handled separately.

P.1.4.4 Gauss-Jackson Predictor

The same analysis can be applied to the Störmer predictor representation, starting with the same comparison between ∇^2 and L^2:

$$\nabla^2 x_{n+1} = h^2 L^2 \ddot{x}_{n+1} = h^2 \left(\sum_{i=0}^{\infty} q_i \nabla^i \ddot{x}_{n+1} \right)$$

Substituting the Störmer predictor coefficients in place of the Störmer-Cowell corrector coefficients (see equation P.25) produces:

$$\nabla^2 x_{n+1} = h^2 \left((q_0 + q_1 \nabla)\, \ddot{x}_{n+1} + \sum_{i=2}^{\infty} q'_i \nabla^i \ddot{x}_n \right)$$

Continuing now with the same analysis, produces:

$$x_{n+1} = h^2 \left(\nabla^{-2}(1-\nabla)\ddot{x}_{n+1} + \sum_{i=0}^{\infty} q'_{i+2} \nabla^i \ddot{x}_n \right)$$

Thus,

$$x_{n+1} = h^2 \left(\nabla^{-2}(\ddot{x}_n) + \sum_{i=0}^{\infty} g'_i \nabla^i (\ddot{x}_n) \right) \tag{P.29}$$

with $g'_i = q'_{i+2}$.

P.1.5 Ordinate Forms of the Integrators

Both the summed-Adams and Gauss-Jackson forms have two sets of components. The first is a power of the summation operator (the inverse backward-difference operator), ∇^{-1}, and the second is the summation over applications of the backward-difference operator.

The first term appears relatively straightforward; it applies to only one historical data value, and can be iteratively traced back, all the way to the epoch. The derivation of that iterative trace is investigated in section *Summation Term* (on page 133). The second term is more complex; at each evaluation step, the backward-difference operator must be applied some number of times to each of the historical data points, up to the desired order of the integrator. An alternative form, the *ordinate* form, handles much of that summation by assigning a new set of coefficients. This is investigated in the section *Ordinate Form Summations* (on the current page).

P.1.6 Ordinate Form Summations

P.1.6.1 Summed-Adams Corrector

We saw earlier (equation P.6) that the powers of the backward-difference operator could be written as

$$\nabla^i x_n = \sum_{j=0}^{i} (-1)^j \binom{i}{j} x_{n-j}$$

Consider the summed-Adams corrector (equation P.26, truncated at the order of the integrator, N:

$$x_n = h\left(\nabla^{-1} + \sum_{i=0}^{N} \sigma_i \nabla^i\right)(\dot{x}_n)$$

The summed term can be expressed as:

$$\sum_{i=0}^{N} \sigma_i \nabla^i (\dot{x}_n) = \sum_{i=0}^{N} \sigma_i \sum_{j=0}^{i} (-1)^j \binom{i}{j} \dot{x}_{n-j} = \sum_{j=0}^{N} \left[(-1)^j \sum_{i=0}^{N} \sigma_i \binom{i}{j}\right] \dot{x}_{n-j}$$

Thus, the summed-Adams corrector coefficients are expressed in ordinate form, with coefficients expressed in terms of the coefficients from the summed-Adams corrector coefficients.

Letting

$$A_{Ci} = (-1)^i \sum_{j=i}^{N} \sigma_j \binom{j}{i} \tag{P.30}$$

the first-order corrector form becomes:

$$x_n = h\left(\nabla^{-1}(\dot{x}_n) + \sum_{i=0}^{N} A_{Ci} \dot{x}_{n-i}\right) \tag{P.31}$$

Note that the new coefficients A_{Ci} do not depend on the previous states, so may be calculated once for any given value N. The use of these coefficients simplifies the application from the multiple application of powers of the ∇ operator to the state $\dot{x}_n$, replacing it with the sum over scaled values of the historical state values, $\dot{x}_{n-i}$.

P.1.6.2 Summed-Adams Predictor

Similarly, the summed term in the summed-Adams predictor expression (equation P.27):

$$x_{n+1} = h\left(\nabla^{-1} + \sum_{i=0}^{\infty} \sigma_i' \nabla^i\right)(\dot{x}_n)$$

can be written as

$$x_{n+1} = h\left(\nabla^{-1}(\dot{x}_n) + \sum_{i=0}^{N} A_{Pi}\dot{x}_{n-i}\right) \tag{P.32}$$

with ordinate coefficients expressed in terms of the coefficients from the summed-Adams predictor coefficients:

$$A_{Pi} = (-1)^i \sum_{j=i}^{N} \sigma_j' \binom{j}{i} \tag{P.33}$$

P.1.6.3 Gauss-Jackson Corrector

For the Gauss-Jackson corrector in ordinate form, we obtain

$$x_n \;=\; h^2\left(\nabla^{-2}(\ddot{x}_{n-1}) + \sum_{i=0}^{N} G_{Ci}(\ddot{x}_{n-i})\right) \tag{P.34}$$

with

$$G_{Ci} = (-1)^i \sum_{j=i}^{N} g_j \binom{j}{i} \tag{P.35}$$

P.1.6.4 Gauss-Jackson Predictor

For the Gauss-Jackson predictor in ordinate form, we obtain

$$x_{n+1} \;=\; h^2\left(\nabla^{-2}(\ddot{x}_n) + \sum_{i=0}^{N} G_{Pi}(\ddot{x}_{n-i})\right) \tag{P.36}$$

with

$$G_{Pi} = (-1)^i \sum_{j=i}^{N} g_j' \binom{j}{i} \tag{P.37}$$

P.1.7 Generation of the Summation Term

P.1.7.1 Adams-sum

The first order summation term is referred to in the code as the *adams_sum.*

Recall that the backward difference operator is defined as $\nabla x_n = x_n - x_{n-1}$. Therefore, the inverse can be found recursively:

$$\begin{aligned} x_n &= \nabla^{-1}x_n - \nabla^{-1}x_{n-1} \ \Rightarrow \\ \nabla^{-1}x_n &= x_n + \nabla^{-1}x_{n-1} \\ &= x_n + x_{n-1} + x_{n-2} + ... + x_1 + \nabla^{-1}x_0 \end{aligned}$$

Hence, the value $\nabla^{-1}x_n$ can be found by iterative summation (hence the name, *summation operator*). The same result can be applied to the derivative terms.

Let the *adams-sum* be defined as

$$\alpha_n \equiv \nabla^{-1}\ddot{x}_n = \alpha_{n-1} + \ddot{x}_n \tag{P.38}$$

It remains only to identify the value of the summation operator applied to the epoch, $\nabla^{-1}\ddot{x}_0$ in order to find the value of the summation operator applied to any point in the future.

Consider the summed-Adams corrector, expressed in ordinate form at the epoch, applied to the first-derivative (as it is in this integration algorithm).

$$\dot{x}_0 = h\left(\nabla^{-1}\ddot{x}_0 + \sum_{i=0}^{N} A_{Ci}(\ddot{x}_{-i})\right)$$

With an evaluation of the summed-term, and knowledge of the epoch state, the epoch summation term can be evaluated.

However, this form is not particularly useful, because it requires *N+1* pre-epoch second-derivative values, which we do not have. We do, however, have *N+1* points around epoch (i.e. the priming points). Suppose there are m post-epoch priming points and *N-m* pre-epoch priming points (typically, for N even, $m = \frac{N}{2}$, and for N odd, $m = \frac{N+1}{2}$); these can all be accessed by very careful applications of the displacement operator, E (recall that $E^{-1} = 1 - \nabla$).

A trivial application to the summed-term in its current truncated form yields:

$$\sum_{i=0}^{N} A_{Ci}(\ddot{x}_{-i}) = \sum_{i=0}^{N} A_{Ci}(E^{-m}E^{m}(\ddot{x}_{-i})) = \sum_{i=0}^{N} A_{Ci}(1-\nabla)^{m}(\ddot{x}_{m-i})$$

In this form, we are using only the known values of the derivative, but have incorporated additional difference operators, which have pushed the integrator beyond order N, and done so inconsistently – the term $\ddot{x}_{m-N-1}$ has been eliminated from direct contribution by truncation of the sum, but indirectly included with terms such as $\nabla\ddot{x}_{m-N}$, and $\nabla^{m}\ddot{x}_{2m-N-1}$.

Correctly truncating the series at N requires that we go back to the regular form.

Consider the same application of the displacement operator to the regular form (equation P.26)

$$\dot{x}_0 = h\left(\nabla^{-1} + \sum_{i=0}^{\infty} \sigma_i \nabla^i\right)(\ddot{x}_0) = h\left(\nabla^{-1}\ddot{x}_0 + \sum_{i=0}^{\infty} \sigma_i (1-\nabla)^m \nabla^i (\ddot{x}_m)\right)$$

The summed-term can be written in its binomial expansion:

$$\sum_{i=0}^{\infty} \sigma_i (1-\nabla)^m \nabla^i = \sum_{i=0}^{\infty} \sigma_i \sum_{j=0}^{m} (-1)^j \binom{m}{j} \nabla^{j+i}$$

Now we can more accurately truncate the series at N by truncating $j+i \leqslant N$, thereby recognizing that the index j is additionally constrained by $j \leqslant N-i$.

Thus,

$$\sum_{i=0}^{\infty} \sigma_i (1-\nabla)^m \nabla^i \approx \sum_{i=0}^{N} \sigma_i \left(\sum_{j=0}^{min(m,(N-i))} (-1)^j \binom{m}{j} \right) \nabla^{i+j}$$

Notice that this covers every combination, subject to the constraints:

$$(0 \leqslant j \leqslant N-i) \; and \; (0 \leqslant j \leqslant m) \; and \; (0 \leqslant i \leqslant N)$$

Notice that $i \leqslant N$ is made redundant by the constraints, $(i+j \leqslant N)$ and $(0 \leqslant j)$.

Letting $k = i + j$, then we have:

$$\begin{aligned} i \geqslant 0 \quad &\Rightarrow \quad 0 \leqslant i+j = k \\ i \geqslant 0 \quad &\Rightarrow \quad j \leqslant i+j = k \\ j \leqslant N-i \quad &\Rightarrow \quad k = i+j \leqslant N \end{aligned}$$

and the constraints become:

$$(0 \leqslant k \leqslant N) \; and \; (0 \leqslant j \leqslant m) \; and \; (j \leqslant k)$$

Subject to these constraints, the sum can be regrouped:

$$\sum_{i=0}^{N} \sigma_i \left(\sum_{j=0}^{min(m,(N-i))} (-1)^j \binom{m}{j} \right) \nabla^{i+j} = \sum_{k=0}^{N} \sigma_{k-j} \left(\sum_{j=0}^{min(m,k)} (-1)^j \binom{m}{j} \right) \nabla^k$$

Thus,

$$\sum_{i=0}^{\infty} \sigma_i (1-\nabla)^m \nabla^i \approx \sum_{i=0}^{N} \left(\kappa_{mi} \nabla^i \right)$$

with

$$\kappa_{mi} \equiv \sum_{j=0}^{min(m,i)} (-1)^j \binom{m}{j} \sigma_{i-j}$$

The correctly truncated form is then:

$$\dot{x}_0 \approx h \left(\nabla^{-1} \ddot{x}_0 + \sum_{i=0}^{N} \kappa_{mi} \nabla^i \ddot{x}_m \right)$$

Now performing the translation to ordinate form of the correctly truncated version produces:

$$\dot{x}_0 = h \left(\nabla^{-1} (\ddot{x}_0) + \sum_{i=0}^{N} A_{Ii} \ddot{x}_{m-i} \right) \tag{P.39}$$

where

$$A_{Ii} = (-1)^i \sum_{j=i}^{N} \kappa_{mj} \binom{j}{i} \tag{P.40}$$

From here, it is easy to obtain the expression for the initial term in the *adams_sum* series, given a complete set of priming points.

The adams-sum,

$$\alpha_n \equiv \nabla^{-1} \ddot{x}_n = \ddot{x}_n + \alpha_{n-1} = \ddot{x}_n + \ddot{x}_{n-1} + ... + \ddot{x}_1 + \alpha_0 \tag{P.41}$$

with

$$\alpha_0 \equiv \nabla^{-1} \ddot{x}_0 = \frac{\dot{x}_0}{h} - \sum_{i=0}^{N} A_{Ii} \ddot{x}_{m-i}$$

(by equation P.39)

P.1.7.2 Gauss-sum

The second-order summation term is referred to in the code as the *gauss_sum*. The derivation is a little more complicated, but follows a similar process.

Let the *gauss-sum*

$$\gamma_n \equiv \nabla^{-2} \ddot{x}_n \tag{P.42}$$

Inverting the double application of ∇

$$\nabla^2 x_n = x_n - 2x_{n-1} + x_{n-2}$$

and then expanding each term yields:

$$\nabla^{-2}x_n = x_n + 2\nabla^{-2}x_{n-1} - \nabla^{-2}x_{n-2} \tag{P.43}$$

Write $p_n = \nabla^{-2}x_n - \nabla^{-2}x_{n-1}$; then by simple substitution:

$$\nabla^{-2}x_n = x_n + \nabla^{-2}x_{n-1} + p_{n-1}$$

Also, since $\nabla^{-1}x_n = x_n + x_{n-1} + ... + \nabla^{-1}x_0$

$$\begin{aligned} p_n &= \nabla^{-1}(\nabla^{-1}x_n) - \nabla^{-1}(\nabla^{-1}x_{n-1}) \\ &= \nabla^{-1}(x_n + x_{n-1} + ... + \nabla^{-1}x_0) - \nabla^{-1}(x_{n-1} + x_{n-2} + ... + \nabla^{-1}x_0) \\ &= \nabla^{-1}x_n \end{aligned}$$

Hence,

$$\nabla^{-2}x_n = x_n + \nabla^{-2}x_{n-1} + \nabla^{-1}x_{n-1} \tag{P.44}$$

Thus,

$$\gamma_n = \ddot{x}_n + \gamma_{n-1} + \alpha_{n-1} \tag{P.45}$$

(i.e. the sum of this term, the previous iteration of the *gauss-sum* term, and the previous iteration of the *adams-sum* term, making the gauss-sum evaluation intrinsically dependant upon the adams-sum evaluation).

Note that there are inevitably two constant terms in this expression, ultimately γ_0 and α_0. The latter of these we have already found; the former can be determined using the same methods.

Consider the evaluation at epoch:

$$x_0 = h^2 \left(\nabla^{-2}\ddot{x}_{-1} + \sum_{i=0}^{N} g_i (1-\nabla)^m \nabla^i (\ddot{x}_m) \right)$$

Combining equations P.43 and P.44

$$\nabla^{-2}x_1 = x_1 + 2\nabla^{-2}x_0 - \nabla^{-2}x_{-1} = x_1 + \nabla^{-2}x_0 + \nabla^{-1}x_0$$

yields

$$\nabla^{-2}x_{-1} = \nabla^{-2}x_0 - \nabla^{-1}x_0$$

Hence,

$$x_0 = h^2 \left(\gamma_0 - \alpha_0 + \sum_{i=0}^{N} g_i (1-\nabla)^m \nabla^i (\ddot{x}_m) \right)$$

Following the same method as for the *adams-sum*, the summed term can be written in ordinate form:

$$\sum_{i=0}^{N} g_i (1-\nabla)^m \nabla^i = \sum_{i=0}^{N} (\lambda_{mi}) \nabla^i$$

with

$$\lambda_{mi} \equiv \sum_{j=0}^{min(m,i)} (-1)^j \binom{m}{j} g_{i-j}$$

Then:

$$x_0 = h^2 \left(\gamma_0 - \alpha_0 + \sum_{i=0}^{N} G_{Ii} \ddot{x}_{m-i} \right)$$

where

$$G_{Ii} = (-1)^i \sum_{j=i}^{N} \lambda_{mj} \binom{j}{i} \tag{P.46}$$

Now, with knowledge of $\alpha_0 = \nabla^{-1}\ddot{x}_0$ from the *adams_sum*, it is clear that

$$\gamma_0 = \nabla^{-2}\ddot{x}_0 = \frac{x_0}{h^2} + \nabla^{-1}(\ddot{x}_0) - \sum_{i=0}^{N} G_{Ii} \ddot{x}_{m-i} \tag{P.47}$$

P.2 Putting it together

Consider the current state, *n*, moving to the next state, *n+1*. This next state must first be predicted and then corrected for the determination of both the first-derivative state (e.g. velocity) using the Summed-Adams formulation, and for the zeroth-derivative state (e.g. position) using the Gauss-Jackson formulation.

P.2.1 Summed-Adams

- Predictor:

$$\dot{x}_{n+1} = h \left(\alpha_n + \sum_{i=0}^{N} A_{Pi}\ \ddot{x}_{n-i} \right) \tag{P.48}$$

- Corrector:

$$\dot{x}_{n+1} = h \left(\alpha_{n+1} + \sum_{i=0}^{N} A_{Ci}\ \ddot{x}_{n+1-i} \right) \tag{P.49}$$

With the *adams-sum* defined as:

$$\alpha_n = \alpha_{n-1} + \ddot{x}_n$$

and

$$\alpha_0 = \frac{\dot{x}_0}{h} - \sum_{i=0}^{N} A_{Ii} \ddot{x}_{m-i}$$

The corrector term must be repeatedly evaluated and corrected as a function of the second derivative of the state being calculated ($\ddot{x}_{n+1}$). It therefore makes sense to divide this into two parts, one that is already determined, and one that is variable.

(Note that $\alpha_{n+1} = \alpha_n + \ddot{x}_{n+1}$ is naturally separated into known and unknown parts, also).

$$\dot{x}_{n+1} = h\left(\left(\alpha_n + \sum_{i=1}^{N} A_{Ci}\ddot{x}_{(n+1)-i}\right) + (1 + A_{C0})\ddot{x}_{n+1}\right) \tag{P.50}$$

By the time the corrector phase is entered, the derivative history will include $\ddot{x}_{n+1}$ as the most recent back-point (stored at position 0 in the array). Position 1 will give $\ddot{x}_n$, position 2 $\ddot{x}_{n-1}$, etc. Hence, both A_{Ci} and $\ddot{x}_{(n+1)-i}$ are at position i in their respective arrays.

Note – because this is the only place that the coefficient A_{C0} is used, the corresponding value in the code, *sa_corrector_coeff[0]* is given the value $A_{C0} + 1$. All other elements *sa_corrector_coeff[i]* are given the value A_{Ci}.

P.2.2 Gauss-Jackson

- Predictor:

$$x_{n+1} = h^2\left(\gamma_n + \sum_{i=0}^{N} G_{Pi}(\ddot{x}_{n-i})\right) \tag{P.51}$$

- Corrector:

$$x_{n+1} = h^2\left(\gamma_n + \sum_{i=0}^{N} G_{Ci}(\ddot{x}_{n+1-i})\right) \tag{P.52}$$

With the *gauss-sum* defined as:

$$\gamma_n = \gamma_{n-1} + \alpha_{n-1} + \ddot{x}_n$$

and

$$\gamma_0 = \frac{x_0}{h^2} + \alpha_0 - \sum_{i=0}^{N} G_{Ii}\ddot{x}_{m-i}$$

Similarly, this corrector term can be broken into a determined and a variable component:

$$x_{n+1} = h^2\left(\left(\gamma_n + \sum_{i=1}^{N} G_{Ci}(\ddot{x}_{(n+1)-i})\right) + G_{C0}(\ddot{x}_{n+1})\right)$$

with both G_{Ci} and $\ddot{x}_{(n+1)-i}$ at position i in their respective arrays.

P.3 Validation of Gauss-Jackson Coefficients

The coefficients generated by this algorithm were independently validated against those for the 8^{th} order Gauss-Jackson algorithm presented by Berry and Healy [2].

1. All *predictor* coefficients match their analogous values presented in Berry and Healy.
2. All but one of the *corrector* coefficients match; the exception is the first element in the summed-adams corrector.

Berry and Healy present the analogous b_{44}, used as:

$$r_n = h\left(\nabla^{-1}\dot{r}_n - \frac{\dot{r}_n}{2} + b_{44}\dot{r}_n + \sum_{k=-4}^{3} b_{4k}(\dot{r}_{n+k-4})\right)$$

(see equations (68), (77))

We have, from equation P.31:

$$x_n = h\left(\nabla^{-1}\dot{x}_n + A_{C0}\dot{x}_n + \sum_{i=1}^{N} A_{Ci}\dot{x}_{n-i})\right)$$

(which, with $i = 4 - k$, has identical form)

Consequently, we should expect: $A_{C0} = b_{44} - \frac{1}{2}$,

Recalling from section P.2.1 (see, in particular, equation P.50) that the stored value for the first element of the summed-adams predictor coefficients is $1 + A_{C0}$ rather than A_{C0}, we should expect to see that the stored value

$$1 + A_{C0} = b_{44} + \frac{1}{2} = -\frac{63887}{89600}$$

which is observed.

All but one of the *initializ* coefficients match with their analogous values presented in Berry and Healy; the exception is the fifth element in the summed-adams corrector, *sa_initializ_coeff[4]*.

Berry and Healy present the analogous b_{00}, used as:

$$\dot{r}_0 = h\left(\nabla^{-1}\ddot{r}_0 - \frac{\ddot{r}_0}{2} + \sum_{k=-4}^{-1} b_{0k}\ddot{r}_k + b_{00}\ddot{r}_0 + \sum_{k=1}^{4} b_{0k}\ddot{r}_k\right)$$

(see equation (69))

We have, from equation P.39, recognizing that order $N = 8$ and that therefore the pre-/post- epoch split value $m = 4$:

$$\dot{x}_0 = h\left(\nabla^{-1}\ddot{x}_0 + \sum_{i=0}^{3} A_{Ii}\ddot{x}_{4-i} + A_{I4}\ddot{x}_0 + \sum_{i=4}^{8} A_{Ii}\ddot{x}_{4-i}\right)$$

(again, with $i = 4 - k$, these have identical form)

Consequently, we should expect:

$$A_{I4} = b_{00} - \frac{1}{2} = -\frac{1}{2}$$

which is observed.

Appendix Q

Livermore Solver for Ordinary Differential Equations (LSODE)

Q.1 Overview

The Livermore Solver for Ordinary Differential Equations (LSODE) is an adaptive step integrator. It computes its preferred integration step internally, independently of the step-size requested by the simulation engine. In order to obtain the state at the externally-defined (i.e. by the simulation engine) time-step boundary, LSODE performs an iteration using the two states (at the internally-generated time-step boundaries) on either side of the requested time.

This LSODE implementation is adapted from the fortran DLSODE package [9] (local copy provided for reference). Cross-references to the original code are included as comments in the code throughout this implementation.

The adaptation is driven by two key requirements:

1. Translation from Fortran to C++
2. Architectural transformation to work from within a simulation engine (e.g. Trick).

The latter of these requirements deserves some elaboration to illustrate the fundamentally different paradigms under which this implementation and the original DLSODE implementation operate.

- The DLSODE fortran code [9] is used as a standalone routine. In such a paradigm, the user calls the integrator, then external calls are made – from within the integrator – to compute derivatives along the integration path. The variables are then integrated according to incoming derivative values. The code exits when the simulation has completed to the very end.
- JEOD integration operates as a component of a simulation-engine system. In this paradigm, the simulation engine controls the flow, and the integrator is just one component, called to integrate a specific variable (or collection of variables) for some subset of the integration path. The simulation engine also controls the generation of the derivative values, and feeds those

into the integrator. Consequently, the integrator is typically called many times during a single simulation.

Q.2 Classification

LSODE is a multistep, multi-stage, multi-cycle integrator for vector-valued first order initial value problems. LSODE may use derivatives from previous integration cycles to advance state during the current integration cycle, which makes this method a multistep integrator. LSODE may take multiple internal steps to complete an integration cycle, which makes this method a multi-stage integrator. LSODE may split an integration interval into multiple cycles, which makes this method a multi-cycle integrator.

LSODE explicitly addresses the vector-valued nature of an initial value problem. As such, this technique is more attuned to the geometry of the problem than is the typical scalar-valued solver that treats a vector-valued problem as a set of independent scalar-valued problems.

Q.3 Implemented and Provided Techniques

Table Q.1 lists the types of initial value problems addressed by the ER7 Utilities integration suite that can be used in conjunction with the LSODE technique. As noted in the table, LSODE only implements integrators for a first order problem and a simple second order problems. It does not implement either of the generalized second order problems, and it does not provide alternatives to these unimplemented problems.

Table Q.1: LSODE Initial Value Problems

Problem Type	**Support**
FirstOrderODE	Implemented
SimpleSecondOrderODE	Implemented
GeneralizedDerivSecondOrderODE	Not provided
GeneralizedStepSecondOrderODE	Not provided

Q.4 Mathematical Description

The underlying mathematical model of LSODE has not been appreciably changed by this implementation. For details on the mathematical model, see Radhakrishnan and Hindmarsh [12] (a local copy is provided for reference). The original LSODE package is very versatile, with optional available that make it suitable for the integration of a wide range of dynamic events and processes. This implementation, being specifically intended for integration of orbital dynamics equations, provides a significantly pared-down subset of the original LSODE package, and provides no new capabilities.

The sequence of computations used in each pass through the LSODE integration method depends on the state of the problem at any given time. The original LSODE code [9] was divided into two primary routines (DLSODE and DSTODE) and several subroutines. Internally, each of the two primary subroutines was further subdivided into multiple 'Blocks'.

In this implementation, the subroutines have been divided into separate methods. The two main routines have been further subdivided to reduce the size and complexity of individual methods and to improve navigation within the algorithm. This latter subdivision approximately follows the 'Block' subdivisions of the original code, with additional subdivision where necessary.

Flow charts are provided below to assist with following the LSODE method through one complete integration step.

In the flow charts, file locations are provided to allow the reader to identify the code associated with each step. The integration code is all found in the LsodeFirstOrderODEIntegrator class (note that second-order ODEs are solved as a pair of first-order ODEs; hence they utilize the same code). Because of its size, the code for the first-order ODE integrator is divided into four separate files:

- lsode_first_order_integrator__integrator.cc.
  - This file represents most of the code taken from the DSTODE function.
- lsode_first_order_integrator__manager.cc
  - This file represents the code from the main DLSODE function
- lsode_first_order_integrator__support.cc
  - This file represents the code from the various support functions that had been written or imported to support peripheral operations. These methods tend to be tailored specifically to the LSODE algorithm
- lsode_first_order_integrator__utility.cc
  - This file represents the code for the C++ class itself (e.g. constructor, destructor, memory management), and mathematical algorithms of general applicability.

Thus, when the flow chart refers to *foi_manager*, it is referring to the *lsode_first_order_integrator__manager.cc* file. Line numbers are approximate.

The first six charts represent entry-paths into the integrator, and the last eight represent internal calls. As discussed earlier, it was necessary to divide the LSODE code at any place where external calls were being made. This allows the integrator to completely exit back to the simulation engine, receive the new inputs (that would previously have come from calls to external routines), and re-enter the code back at the appropriate place. The six charts representing entry-paths are uniquely associated with a particular phase of the integration cycle; in the code, the variable *re_entry_point* is used to select the appropriate flow-path. The first six charts are therefore identified with the six possible values of *re_entry_point*:

1. **CycleStartFinish** is used at the start of every integration cycle (a cycle being the commanded value from the simulation engine – integrate from here to there). This value is also used to indicate completion of an integration step. The cycle is recognized as completed only when the integrator returns to the simulation engine with *re_entry_point = CycleStartFinish.*

2. **InitCalc** is used for initializing the integrator at the start of any particular problem or simulation.

3. **JacobianPrep** is used when the integrator builds a Jacobian matrix to evaluate the integration. This mode is used to prepare that Jacobian whenever it needs recomputing.

4. **ResetIterLoop** is used to reset the iteration loop. The integrator basically works under a predict-correct-correct-... paradigm, continuing to correct until convergence. The convergence tests are reset following every predict phase. This mode is used to reset the iteration loop that drives the corrections.

5. **IterationLoop** is used while the integrator is performing its correction iterations, either to convergence or a recognized failure. Convergence leads to the end of a cycle (*CycleStartFinish*) while failure leads to resetting the iteration (*ResetIterLoop*).

6. **DstodeResetStep** is used to reset the time-step. Under some circumstances it is necessary to reset the integration step size (note that LSODE runs on an internally identified step size and interpolates the result to hit the time demanded by the simulation engine).

The general format is the same for all diagrams:

- Dotted line – an optional path. The required conditions are specified on the path.
- Colored line – choice of paths. Again, the conditions are specified on one or all of the paths.
- Text boxes – method calls. The general flow is from top to bottom. Sequential calls from one method to a series of other methods are identified by branching from top to bottom in the sequential order. Each text box contains a method name, an approximate location from where it is called, and an approximate location where it is found. Some boxes are colored to facilitate navigation from module-to-module.
- Plain blue boxes – branching points.

These charts can be used to confirm that there are no closed loops. All entry points (charts 1-6) ultimately result in either the FAIL box on chart 14 or one of the CYCLE COMPLETE boxes. Even in apparent self-calling loops, such as in charts 12-14 (12 calls 13, which calls 14, which calls 12) these loops are resolved when variables such as *internal_state* are used to provide additional flow control.

Q.5 First Order ODE

All LSODE integrations are of first-order equations. The state vector is integrated as a single entity, with all components taking the same time-step and being tested for convergence together.

The LSODE package provides a variable-order, variable-step first-order integrator. As each problem initializes, the order starts at one (1) and the step starts small. As a history is accumulated, both the order and step-size are gradually increased automatically. The order reaches a maximum of 5 or 12, depending on the options chosen; the step-size has no enforced upper bound.

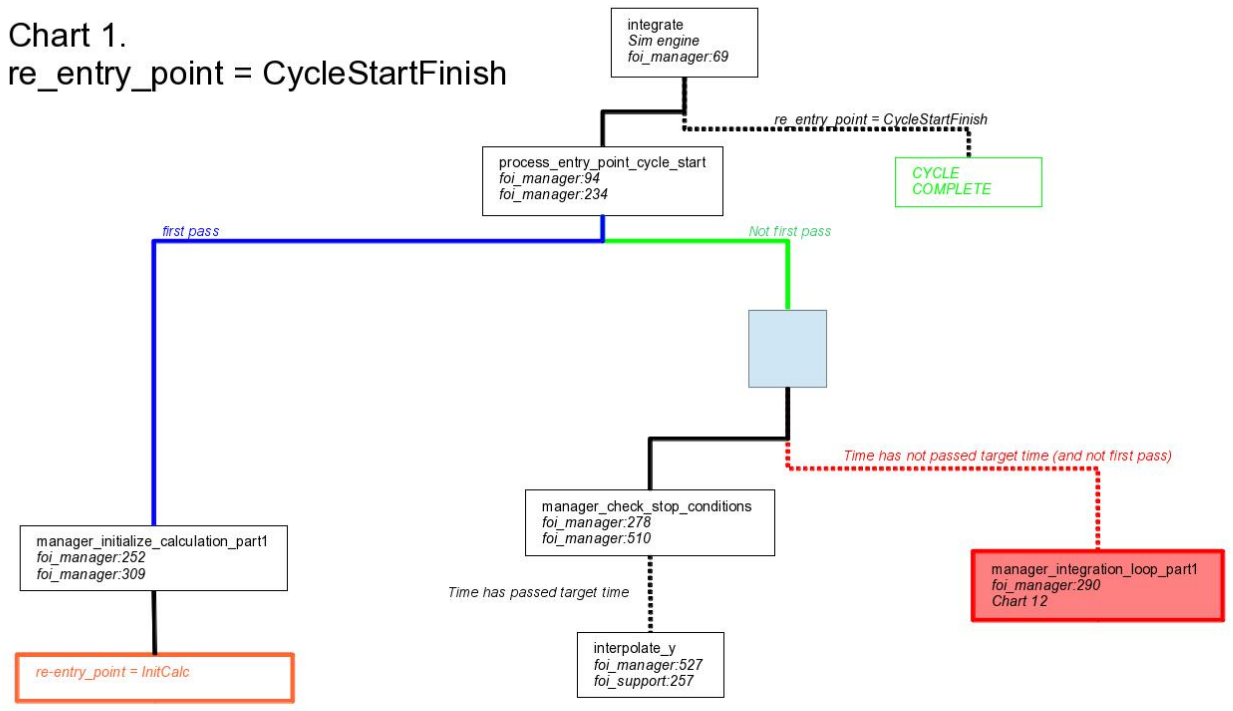


Chart 1.
re_entry_point = CycleStartFinish
integrate
Sim engine
foi_manager:69
re_entry_point = CycleStartFinish
CYCLE COMPLETE
process_entry_point_cycle_start
foi_manager:94
foi_manager:234
first pass
Not first pass
Time has not passed target time (and not first pass)
manager_check_stop_conditions
foi_manager:278
foi_manager:510
manager_integration_loop_part1
foi_manager:290
Chart 12
manager_initialize_calculation_part1
foi_manager:252
foi_manager:309
Time has passed target time
interpolate_y
foi_manager:527
foi_support:257
re-entry_point = InitCalc

Figure Q.1: Flow Chart 1.

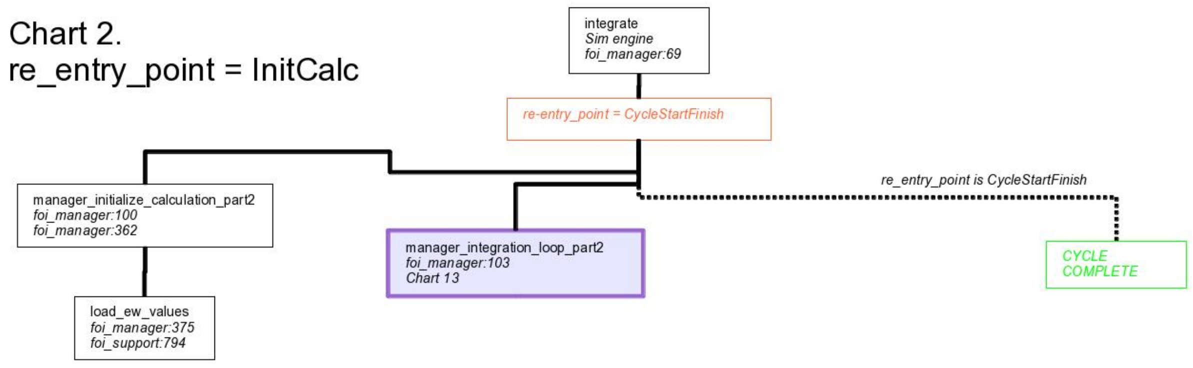


Chart 2.
re_entry_point = InitCalc
integrate
Sim engine
foi_manager:69
re-entry_point = CycleStartFinish
re_entry_point is CycleStartFinish
manager_initialize_calculation_part2
foi_manager:100
foi_manager:362
manager_integration_loop_part2
foi_manager:103
Chart 13
CYCLE COMPLETE
load_ew_values
foi_manager:375
foi_support:794

Figure Q.2: Flow Chart 2.

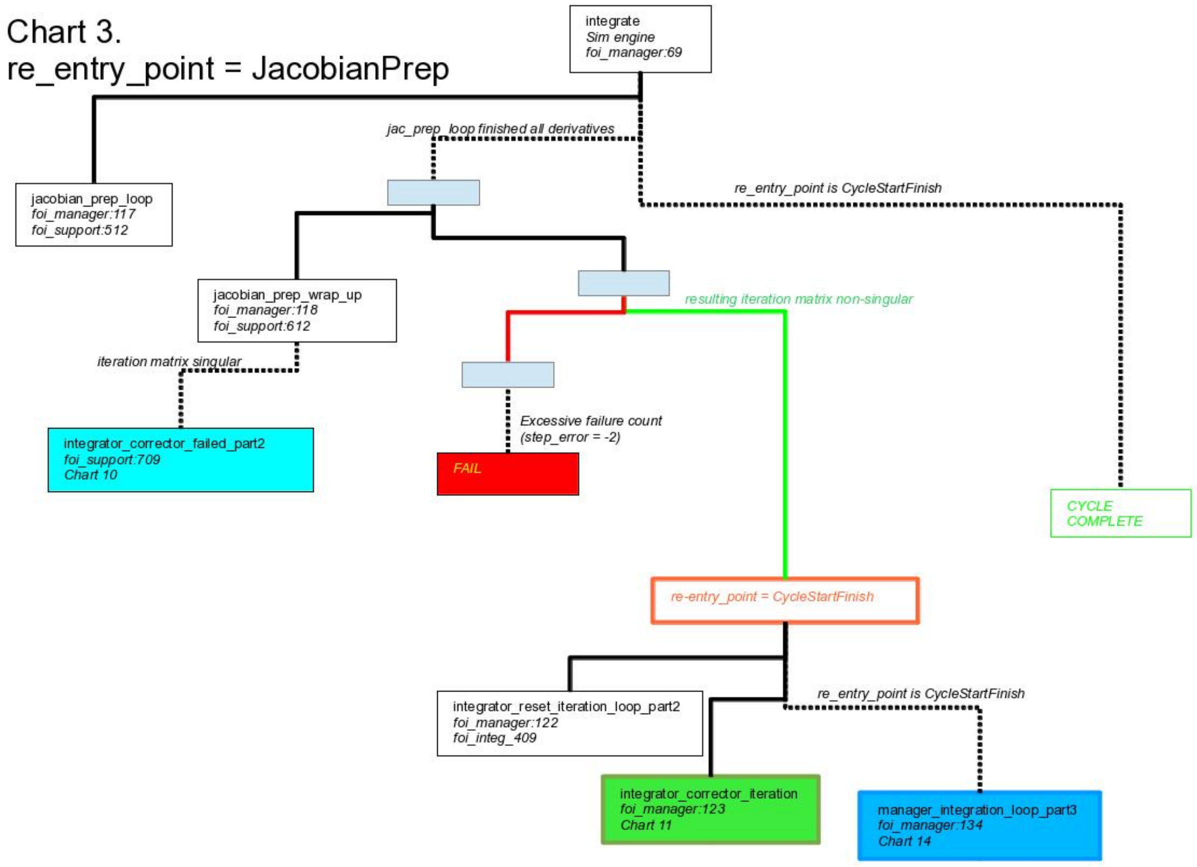


Chart 3.
re_entry_point = JacobianPrep
integrate
Sim engine
foi_manager:69
jac_prep_loop finished all derivatives
re_entry_point is CycleStartFinish
jacobian_prep_loop
foi_manager:117
foi_support:512
jacobian_prep_wrap_up
foi_manager:118
foi_support:612
resulting iteration matrix non-singular
iteration matrix singular
Excessive failure count
(step_error = -2)
integrator_corrector_failed_part2
foi_support:709
Chart 10
FAIL
CYCLE
COMPLETE
re-entry_point = CycleStartFinish
integrator_reset_iteration_loop_part2
foi_manager:122
foi_integ_409
re_entry_point is CycleStartFinish
integrator_corrector_iteration
foi_manager:123
Chart 11
manager_integration_loop_part3
foi_manager:134
Chart 14

Figure Q.3: Flow Chart 3.

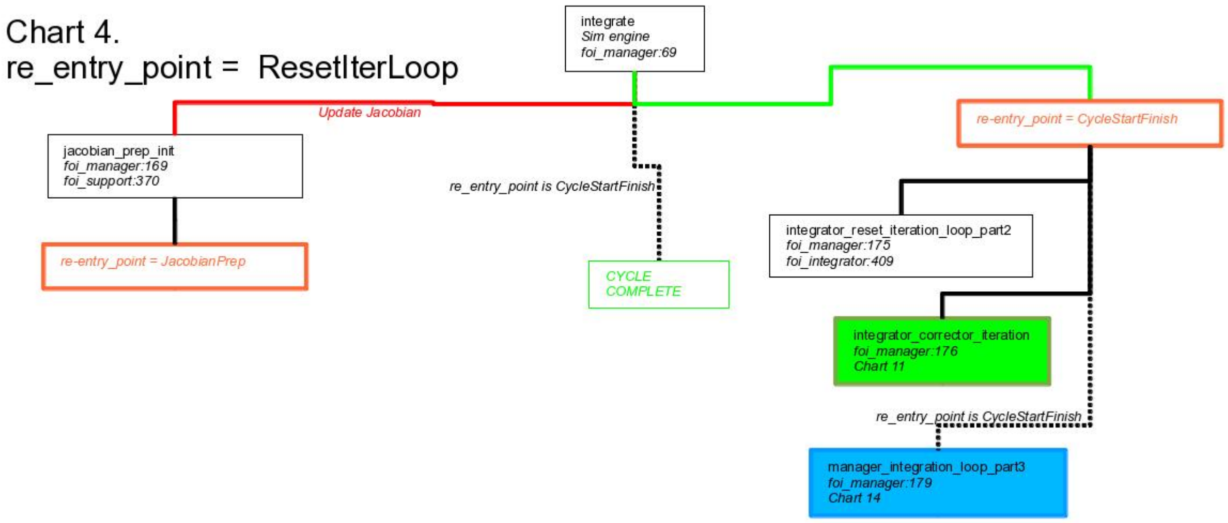


Chart 4.
re_entry_point = ResetIterLoop
integrate
Sim engine
foi_manager:69
Update Jacobian
jacobian_prep_init
foi_manager:169
foi_support:370
re-entry_point = JacobianPrep
re_entry_point is CycleStartFinish
CYCLE
COMPLETE
re-entry_point = CycleStartFinish
integrator_reset_iteration_loop_part2
foi_manager:175
foi_integrator:409
integrator_corrector_iteration
foi_manager:176
Chart 11
re_entry_point is CycleStartFinish
manager_integration_loop_part3
foi_manager:179
Chart 14

Figure Q.4: Flow Chart 4.

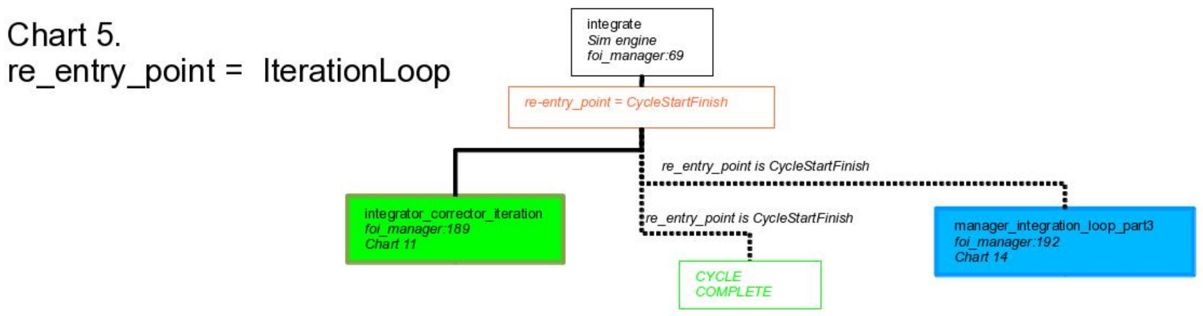


Chart 5.
re_entry_point = IterationLoop
integrate
Sim engine
foi_manager:69
re-entry_point = CycleStartFinish
re_entry_point is CycleStartFinish
integrator_corrector_iteration
foi_manager:189
Chart 11
re_entry_point is CycleStartFinish
CYCLE COMPLETE
manager_integration_loop_part3
foi_manager:192
Chart 14

Figure Q.5: Flow Chart 5.

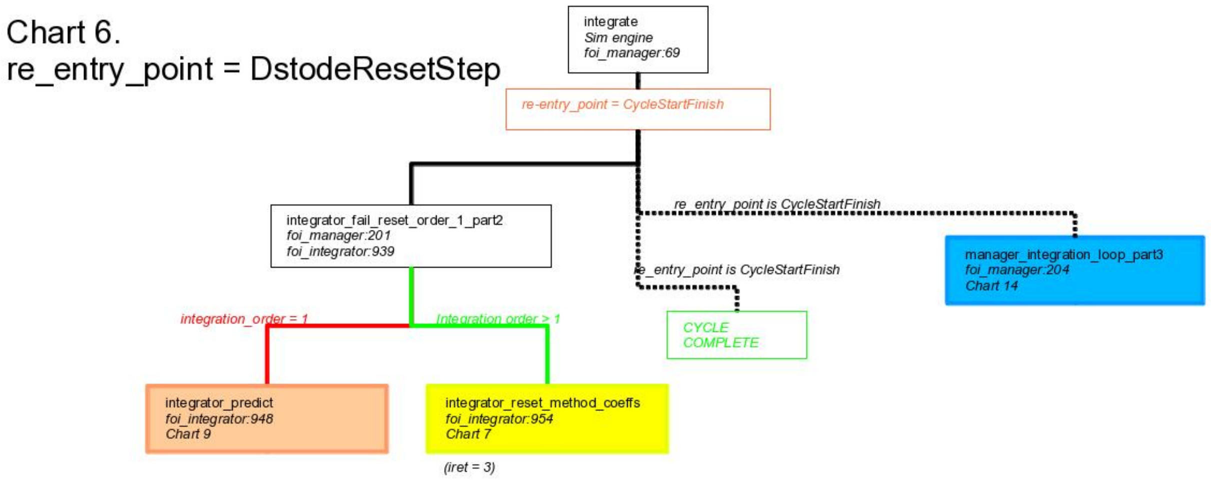


Figure Q.6: Flow Chart 6.

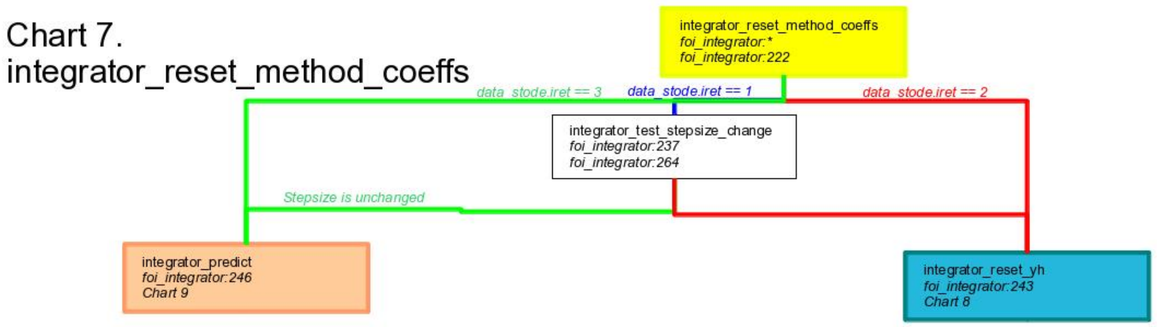


Figure Q.7: Flow Chart 7.

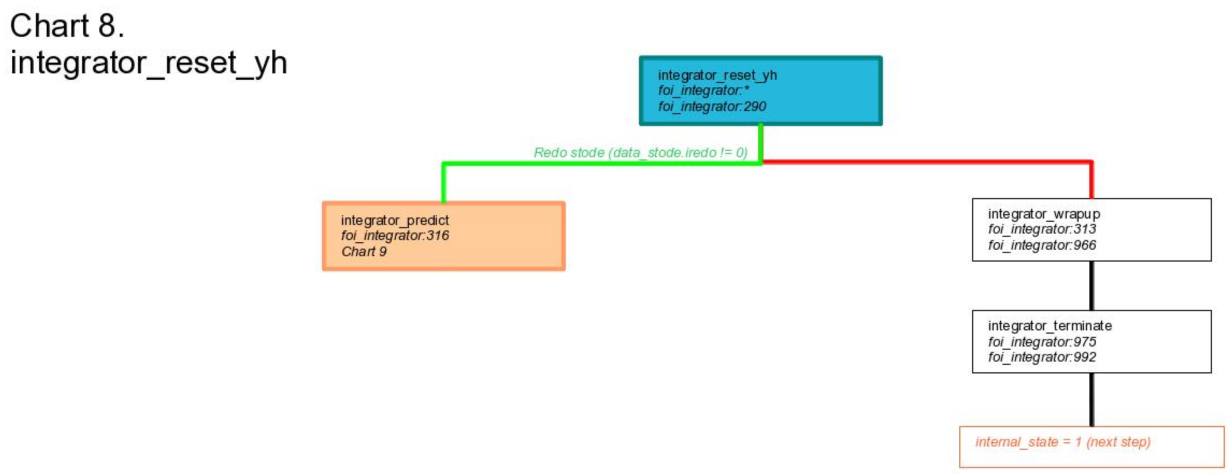


Figure Q.8: Flow Chart 8.

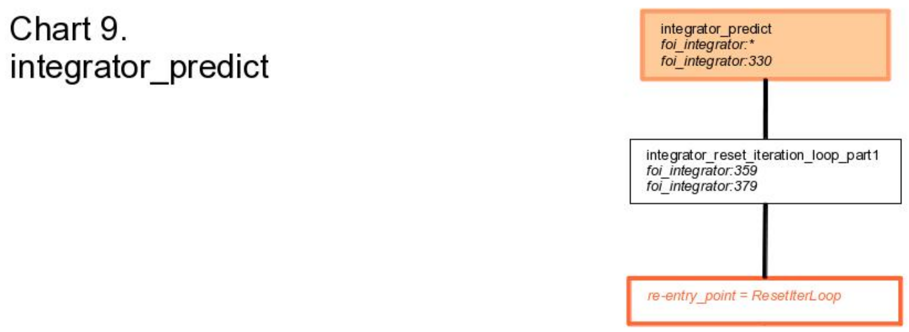


Figure Q.9: Flow Chart 9.

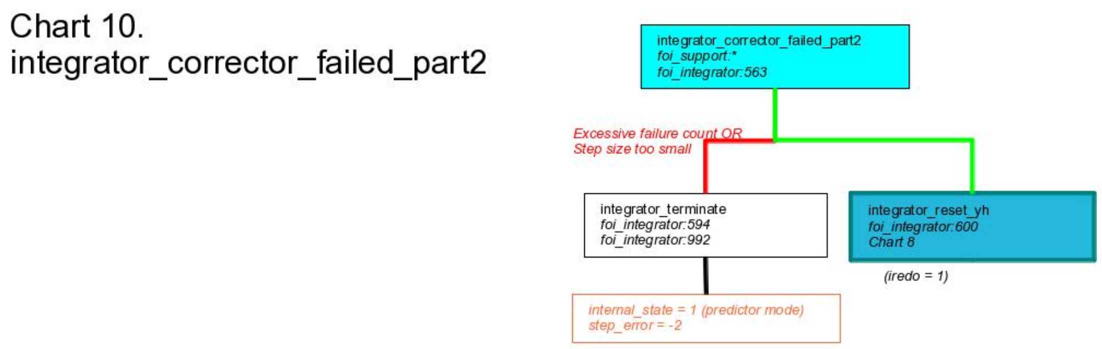


Figure Q.10: Flow Chart 10.

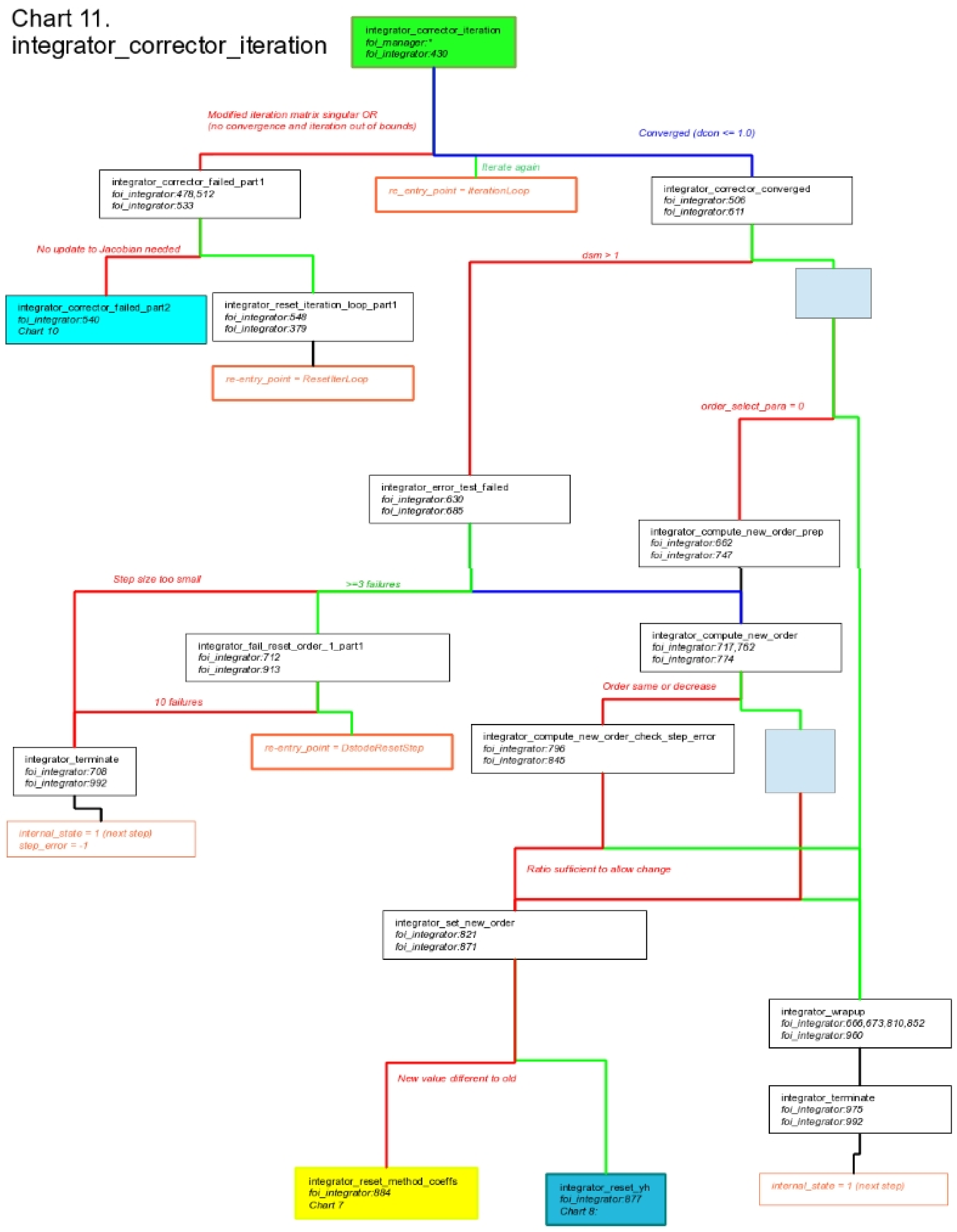


Chart 11.
integrator_corrector_iteration
integrator_corrector_iteration foi_manager:* foi_integrator:430
Modified iteration matrix singular OR (no convergence and iteration out of bounds)
Converged (dcon <= 1.0)
iterate again
integrator_corrector_failed_part1 foi_integrator:478,512 foi_integrator:533
re_entry_point = IterationLoop
integrator_corrector_converged foi_integrator:506 foi_integrator:611
No update to Jacobian needed
dsm > 1
integrator_corrector_failed_part2 foi_integrator:540 Chart 10
integrator_reset_iteration_loop_part1 foi_integrator:548 foi_integrator:379
re-entry_point = ResetIterLoop
order_select_para = 0
integrator_error_test_failed foi_integrator:630 foi_integrator:685
integrator_compute_new_order_prep foi_integrator:662 foi_integrator:747
Step size too small
>=3 failures
integrator_fail_reset_order_1_part1 foi_integrator:712 foi_integrator:913
integrator_compute_new_order foi_integrator:717,762 foi_integrator:774
Order same or decrease
10 failures
integrator_terminate foi_integrator:708 foi_integrator:992
re-entry_point = DstodeResetStep
integrator_compute_new_order_check_step_error foi_integrator:796 foi_integrator:845
internal_state = 1 (next step) step_error = -1
Ratio sufficient to allow change
integrator_set_new_order foi_integrator:821 foi_integrator:871
integrator_wrapup foi_integrator:666,673,810,852 foi_integrator:960
New value different to old
integrator_terminate foi_integrator:975 foi_integrator:992
integrator_reset_method_coeffs foi_integrator:884 Chart 7
integrator_reset_yh foi_integrator:877 Chart 8:
internal_state = 1 (next step)

Figure Q.11: Flow Chart 11.

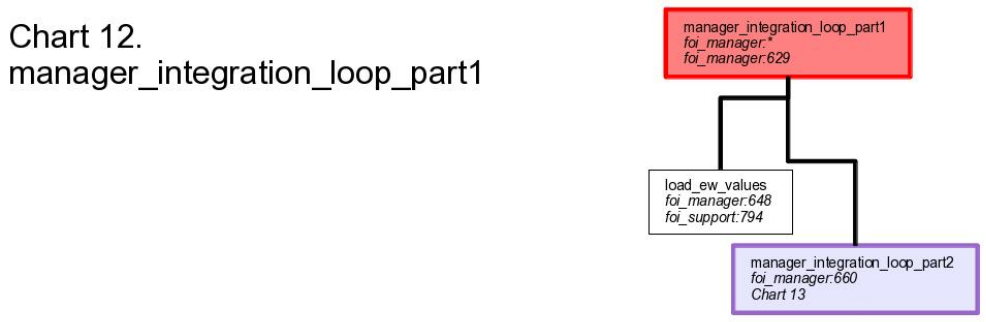


Figure Q.12: Flow Chart 12.

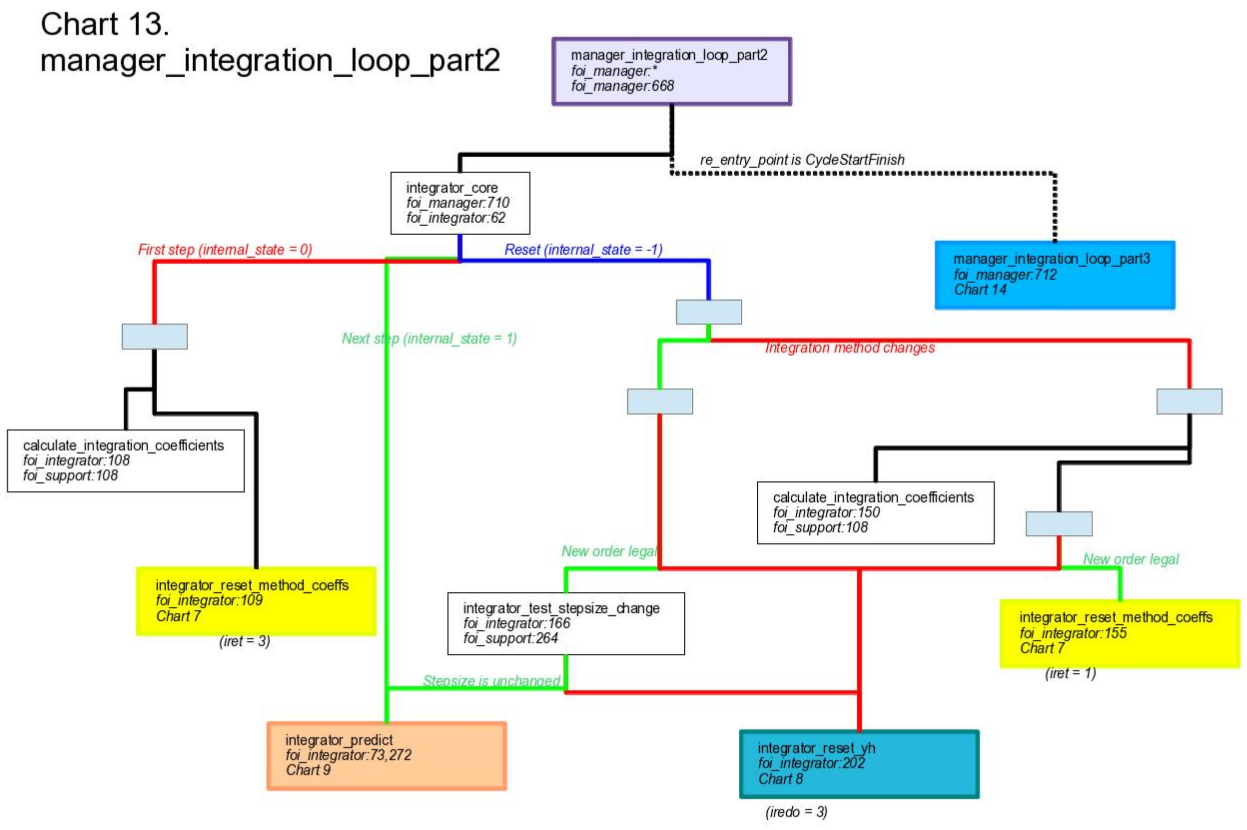


Figure Q.13: Flow Chart 13.

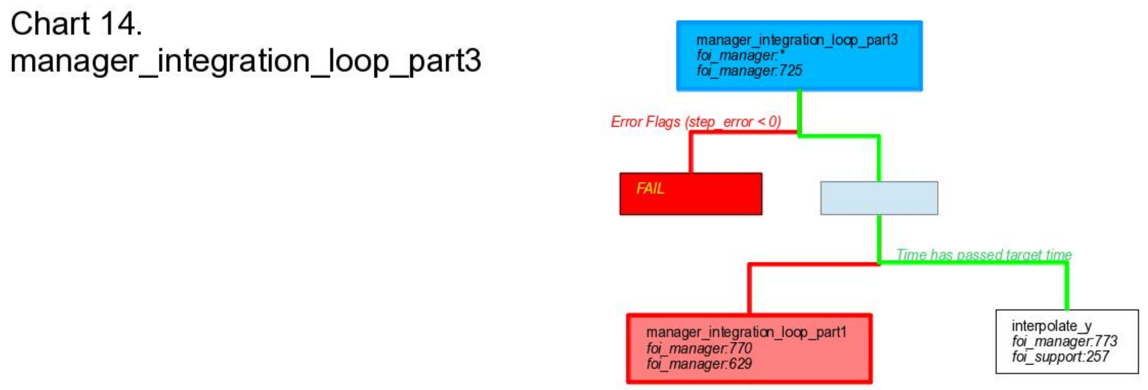


Chart 14.
manager_integration_loop_part3
manager_integration_loop_part3
foi_manager:*
foi_manager:725
Error Flags (step_error < 0)
FAIL
Time has passed target time
manager_integration_loop_part1
foi_manager:770
foi_manager:629
interpolate_y
foi_manager:773
foi_support:257

Figure Q.14: Flow Chart 14.

Unlike the other integrators used in JEOD, the time-step for integration is not enforced from the outside. The LSODE package determines its preferred integration step and integrates to that point. If its integration step exceeds the target time (dictated from the outside, such as by the simulation engine), the solution at that target time is interpolated from the history and the newly created future point. The next integration cycle continues from the end of the previous integration step, not from the interpolated point.

It is possible that the internal step-size could grow to exceed the externally-requested time-step, thereby allowing for the possibility that multiple target times boundaries could be crossed within one internal time-step. In this case, each such target would be iterated from the generated states, a process that continues until the next target time is in the future of the last internally generated time.

Q.6 Second Order ODE

Second-order equations are treated as a pair of coupled first-order equations. The entire state vector (now comprising the zeroth-derivative and first-derivative values) is integrated as one entity.

Q.7 Generalized Position / Generalized Velocity ODE

The LSODE integrator constructor currently does not provide a means to solve a generalized position / generalized velocity second order ODE. As such, it is not suitable for integrating rotational state.

Q.8 Lie Group Second Order ODE

The LSODE integrator constructor currently does not provide a means to solve a Lie group (generalized step) second order ODE.

Q.9 Usage Instructions

Q.9.1 Disable Rotational State

This implementation of LSODE is limited to integrating translational state only. It lacks the ability to handle the generalized-position / generalized-velocity problems associated with integrating rotational state.

To use LSODE to integrate the state of 6-DOF bodies (e.g. bodies of type *DynBody*), the rotational state must be turned off and the *three-dof* flag set to true.

```
Example:
   vehicle.dyn_body.rotational_dynamics = False
   vehicle.dyn_body.three_dof = True
```

Q.9.2 Force Ephemerides Rate High

The Ephemerides Model Manager (most commonly found as the Dynamics Manager) contains a flag, *deriv_ephem_update*, that indicates whether or not the ephemerides update should be performed at the derivative rate (i.e., whether or not the planetary positions should be computed at every point in time corresponding to a computation of the state derivatives).

In most simulations (including those built following recommended practice from the pre-packaged simulation-modules), the ephemerides update is called at the integration rate. For most integrators, this is typically a short timescale, on the order of tens of seconds or less. Consequently, for most scenarios, updating the planetary positions at the derivative rate is entirely unnecessary and the default setting for this flag is *false*.

However, LSODE can handle very large integration steps. Indeed, since it can take many internal steps per specified time-step, the specified (external) time-step is really limited only by the maximum allowable number of internal steps per external step (this is a user-specified value, see below). Relying only on the default ephemerides update to reposition the planets at a pre-determined interval – that may be orders of magnitude larger than necessary – is not a good idea. Therefore, when using LSODE, it is **strongly recommended** that the flag be set.

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

If this step is omitted and the ephemerides updated too sparsely, each planet will physically jump to a new position that is a significant departure from its old position. Consequently, the gravitational accelerations experienced by the vehicle can go through a significant discontinuity. The resulting effect in the integrator is quite subtle and easily overlooked.

The prediction of the next state is made based on the derivative values at the end of the previous internal step, while the correction calculation is made based on the derivative values at the start of the current internal step. When the derivatives are smooth, these values differ minimally and convergence can be quickly achieved over very large time-steps. When there is a discontinuity in the acceleration, the corrector adjustments will not easily converge with the original predictor calculations. LSODE will automatically reduce the internal timestep until they do converge. The effect is that while the integration step may be hundreds of thousands of seconds (or larger), the internal step could be limited to milliseconds (or smaller) to enforce the convergence across the discontinuity. From there, the internal time-step will be allowed to grow, but may only reach tens or hundreds of seconds before the next ephemerides update causes another discontinuity and the process repeats. The end result is that the LSODE integrator will perform very poorly – both in terms of numerical accuracy and speed – but it will not fail in any obvious or recognizable way.

A further incidental complication arises with the interpolation of state to the target time. Suppose the integrator is to integrate from time t_0 to time t_1 and from t_1 to t_2. It will probably overshoot to time $t_1' > t_1$ and interpolate back to the target time t_1. The next phase of integration will begin where the previous left off, at t_1' . Thus, the ephemerides data will change partway through the integration step going from t_1 to t_2. The ephemerides data from t_0 will be used for $t_1 < t < t_1'$ and the data from t_1 will be used for $t_1' < t < t_2$. This invalidates any comparison to other integrators.

By forcing the ephemerides update to the derivative rate (the fastest possible rate given the structure of the integrator), control over when the update is performed passes from the pre-configured schedule of the simulation engine back to the integrator itself. The integrator may, in some cases, be able to handle updates that are thousands of seconds apart, but that determination is better left to the integrator as it computes its own internal step-size.

Q.9.3 Configuring the Integrator

There are several options available for configuring the integrator. These values are found in the *LsodeControlDataInterface* class, instantiated within the integrator constructor at *<lsode_integrator_constructor>.data_interface*.

Two configuration parameters – the relative error tolerance and the absolute error tolerance – must be specified; the others are optional.

As with other integrators, the constructor is only used in the process of creating the integrator; changes made to data elements in the constructor can only be propagated to the integrator when it is constructed. Additional changes made after the integrator has been built will be ignored. Thus, these options are only available at initialization.

For example, for a simulation in which the integrator is specified at the input-file level, one might find:

```
lsode_integrator = trick.LsodeIntegratorConstructor()
lsode_integrator.data_interface.set_rel_tol (0, 1.0E-15)
lsode_integrator.data_interface.set_abs_tol (0, 1.0E-12)
```

Q.9.3.1 Error Tolerance

The allowable error tolerance for an integration process to be considered converged is a combination of two factors - an absolute value (that is independent of the state) and a relative value (that scales linearly with the state). These values are combined to give the total error tolerance.

When subsequent computations produce predictor/corrector values that are within the total error tolerance of one another, the integration has converged.

Ideal values for the absolute and relative error tolerances are problem-specific, and often component-specific. Hence, it is necessary to specify these two values for the particular problem at hand. The default values are illegal.

One or both must be non-zero; if both the absolute and relative tolerances are zero, there is zero permissible deviation, and the integrator can never converge.

Q.9.3.2 Absolute Error Tolerance

abs_tolerance_error_control_vec

- Type: double vector; double array.
- Default: -1.0

The absolute error tolerance provides the permissible difference between subsequent corrector-iterations in order for the corrector to be considered converged. **This value MUST be specified**; the default value is illegal. A zero value means that the absolute error will not be considered in determining the convergence. Initially, the values will populate the vector *abs_tolerance_error_control_vec*. Once complete, these values will be copied into an appropriately-sized allocated array, *abs_tolerance_error_control*. The method *set_abs_tol* should be used to populate these values. This method takes 2 arguments, the index and the value.

```
lsode_integrator.data_interface.set_abs_tol(index,value)
```

Notes:

- No units are specified for this value. The specified value assumes the default units for the variable to which it is being applied. When integrating an orbital body, that is meters for position and meters-per-second for velocity.
- If using a common specification for all absolute tolerances (see Error Control Indicator below), only one value is needed; it must be input at index 0.
- All specified values must be non-negative.
- If an entry is made at index n ($n > 0$) then all non-specified values at prior indices ($i < n$) default to -1.0 and must be specified before continuing.
- If selecting options *SpecificAbs** (see Error Control Indicator below), unspecified entries at indices higher than the highest specified index default to 0.0 and no check is made that values have been input for these components.
- An absolute error tolerance of 0.0 is legal, but dangerous. If one of the state components happens to be extremely close to 0, the relative error will be consequentially small, and the

allowable error may be too small for the machine to handle. Absolute error tolerances should be greater than 0.0 unless the user is absolutely confident that their state component(s) can never go to zero.

- In the absence of a legitimate value, use something small, such as 10^{-12}.

Q.9.3.3 Relative Error Tolerance

rel_tolerance_error_control_vec

- Type:Vector, double; double array.
- Default: -1.0

The relative error tolerance provides the permissible difference between subsequent corrector-iterations as a fraction of the state itself in order for the corrector to be considered converged. **This value MUST be specified**; the default value is illegal. A zero value means that the relative error will not be considered in determining the convergence. Initially, the values will populate the vector *rel_tolerance_error_control_vec*. Once complete, these values will be copied into an appropriately-sized allocated array, *rel_tolerance_error_control*. The method *set_rel_tol* should be used to populate these values. This method takes 2 arguments, the index and the value.

```
lsode_integrator.set_rel_tol(index,value)
```

Notes:

- If using a common specification for all relative tolerances, only one value is needed; it must be input at index 0.
- All specified values must be non-negative.
- If an entry is made at index n ($n > 0$) then all non-specified values at prior indices ($i < n$) default to -1.0 and must be specified before continuing.
- If selecting options **SpecificRel*, (see Error Control Indicator below) unspecified entries at indices higher than the highest specified index default to 0.0 and no check is made that values have been input for these components.
- An absolute error tolerance of 0.0 is legal, but dangerous. If one of the state components happens to grow large, the relative error associated with a small absolute error against a large state may be too small for the machine to handle. For example, an absolute tolerance of 10^{-12} on a state with a value of 10^6 is only one part in 10^{18}, which demands precision that exceeds the capability of a double-precision value. Such an absolute error loses all significance and is effectively zero. Relative error tolerances should be greater than 0.0 unless the user is absolutely confident that the specified absolute tolerance will always be significant across the entire range of the possible state.
- In the absence of a legitimate value, use something close to the limit of resolution of a double precision, such as 10^{-15} (i.e. one part in 10^{15}).

Q.9.3.4 Error Control Indicator

error_control_indicator

- Type: ErrorControlIndicator
- Options:
  1. CommonAbsCommonRel
  2. SpecificAbsCommonRel
  3. CommonAbsSpecificRel
  4. SpecificAbsSpecificRel
- Default: CommonAbsCommonRel (1)

Each integration cycle completes when the corrector-phase produces a result that is within some specified tolerance of the previous corrector result. The tolerances must be specified to be either (or both) relative to the state (e.g. 1 part in 10^{12}) or an absolute discrepancy of default units (e.g. within $10^{-9}m$). These tolerances can also be specified to be applied commonly across all components, or with a specific value for each component. Note that in the case where different state components have different units (e.g. position and velocity in a 6-vector), the numerical value for the absolute error tolerance will be interpreted as being expressed in the default units of the variable (in this case, m and ms^{-1}).

This control flag indicates which of these options to choose. Specification of the tolerance values is covered in the two sections immediately preceding.

1. **CommonAbsCommonRel**
   Apply a common absolute tolerance and common relative tolerance to all components.
2. **SpecificAbsCommonRel**
   Apply a per-axis absolute tolerance and a common relative tolerance.
3. **CommonAbsSpecificRel**
   Apply a common absolute tolerance and a per-axis relative tolerance.
4. **SpecificAbsSpecificRel**
   Apply a per-axis absolute tolerance and another per-axis relative tolerance.

Q.9.3.5 Number of ODEs

num_odes

- Type: unsigned int
- Default: 3, set to 6 for integration of a *Simple6DofDynBody*

This values specifies the number of ODES in the system to integrate. Note that this option is invalid when using LSODE for the JEOD's default purpose of integrating vehicle state. The creation of the integrator for that purpose will automatically assign this value to six (three position and three velocity values).

Q.9.3.6 Integration Method

integration_method

- Type:IntegrationMethod
- Options:
    1. ImplicitAdamsNonStiff
    2. ImplicitBackDiffStiff
- Default:ImplicitAdamsNonStiff

The integration_method determines which integration algorithm to use. For stiff problems, the ImplicitBackDiffStiff option should be chosen. Most orbital problems are not considered stiff and the default to use is the ImplicitAdamsNonStiff. The maximum order is 12 for the ImplicitAdamsNonStiff method, and 5 for the ImplicitBackDiffStiff method.

Q.9.3.7 Corrector Method

corrector_method

- Type:CorrectorMethod
- Options:
    0. FunctionalIteration
    2. NewtonIterInternalJac
    3. JacobiNewtonInternalJac
- Default: FunctionalIteration

The *corrector_method* determines which algorithm to use for the corrector-phase of the integration. FunctionalIteration is the simplest and the default. NewtonIterInternalJac uses a modified Newton iteration scheme utilizing an internally generated numerical Jacobian JacobiNewtonInternalJac uses a modified Jacobi-Newton iteration scheme utilizing an internally generated numerical Jacobian Note that options 1, 4, and 5 (modified Newton iteration with user-supplied Jacobian, with user-supplied banded Jacobian, and with an internally generated banded Jacobian) are not supported in this implementation.

Q.9.3.8 Minimum Step Size

min_step_size

- Type: double
- Default: 0.0

For users for whom keeping the simulation ticking over is more important than precision, there is the option of specifying a minimum step size. Note that this represents the minimum absolute value of the step size; for simulations with negative step sizes, this would still be a positive value (and would represent the least-negative step size). The growth of the step-size is a non-trivial process. When the integrator starts a problem, the step-size is typically small. Every few steps, the step-size is re-evaluated along with the integration order. A step size ratio (new step-size / old step-size) is computed for each of 3 cases:

1. the integration order decreases,
2. the integration order is held constant,
3. the integration order increases.

Whichever order change produces the largest step-size increase is adopted. The corresponding step-ratio is compared against that which would be required to reach the user-specified minimum step-size (this parameter), and the larger of the two selected (step omitted if no minimum step-size is input). Finally, the step-ratio is limited to be no greater than 10.0. The most challenging convergence problems associated with the usage of this parameter appear to occur during integrator start-up. These problems tend to occur when the step-size growth is forced by the minimum step-size parameter and when it is significantly larger than the computed ideal ratio.

For example, consider an initial step-size of 0.1s, a specified minimum of 20.0s, and a computed ratio of 4.2. The next ideal step-size would be 0.42s, the external value of 20.0s is greater, so this takes precedence. The 10.0 limit on the ratio ultimately forces the next time-step to 1.0 s, but that is still significantly larger than the ideal step-size. This may have problems converging. Do not misinterpret this parameter as a preferred step size, particularly for the case of establishing LSODE as a long-arc integrator. While it may be desirable to have a large step size for long-arc integration, it would not be a good idea to specify that value as the minimum step-size. Use this parameter carefully.

Q.9.3.9 Maximum Step Size

max_step_size

- Type: double
- Default: 0.0

For systems with states that are easily predictable very far into the future (e.g. systems with constant gradients) the step sizes can grow extremely large. If it is desirable to maintain the step size closer to the requested step size and avoid extensive interpolation, this option can be used. The default value for this parameter is 0.0, but that is interpreted as infinity. Any non-zero value will be interpreted as specified. This parameter is relatively safe to use but can lead to a lack of accuracy. By forcing the time-step to be somewhat arbitrarily small, more integration steps are taken than necessary and numerical error could be increased. However, the problems with convergence that frequently interfere with setting the minimum step-size are unlikely to be encountered as a result of setting this parameter.

Q.9.3.10 Initial Step Size

initial_step_size

- Type: double
- Default: None

This is the very first time-step to try. If not specified, a value will be computed. There are some overriding safeguards on this value, and the internal error-checking often makes setting this value redundant anyway. If the value is too high for the integrator to converge, the integrator will reduce the step-size automatically until it is suitable for convergence. After convergence, if the integrator fails to meet the error checks, the entire time-step (based on the user-specified value) will be discarded and the integrator will return to its previous safe value; this is normal operating practice. In the case of the first step, the previous safe value is 0.0 and the integrator will generate its own time-step, just as it would have done without this value being set.

Without this value being set, the integrator will err on the side of caution and the initial time-step will tend to be too small. However, it will grow by a factor of 10 at each correction until it reaches a more suitable value, a process that tends to be completed relatively quickly.

Setting this parameter would only be useful if:

- the user knows, a priori, a suitable value for the starting time-step (i.e., a time-step that allows the integrator to converge without error), and
- the specified initial time-step is significantly larger than the value that the integrator would start with, and
- the user is planning on running a large number of simulations using that value where each simulation is very short so that the initialization time is a significant fraction of the overall simulation time.

If any of these conditions are not met, setting this parameter is unlikely to have any noticeable and beneficial effect.

Q.9.3.11 Maximum Order

max_order

- Type: unsigned int
- Default: 12

This is the maximum order allowable for the integrator. The order starts at 1 and generally increases until it reaches either:

- The user-specified maximum order (this value) or
- The maximum-allowable order for the integration method (see *integration_method*)

Specifying a *max_order* value greater than the allowable order for the integration method is completely redundant.

The safe time-step used by a lower order integrator is smaller than that for a higher order integrator. Consequently, setting this value to less than the maximum order for the integration algorithm will result in more steps being taken, slowing down the simulation. Reducing the order is unlikely to have any beneficial performance effects. Unless the user has a specific purpose for limiting the order (e.g. to compare the performance of two 8th order integrators), this parameter should probably not be set.

Q.9.3.12 Maximum Number of Small-step Warnings

max_num_small_step_warnings

- Type: unsigned int
- Default:10

Occasionally, the integrator may get into a situation in which the step size it can take is so small that it is effectively zero. In this case, the integrator will back up to a previous state and try again. On each of these occassions, a warning will be sent to the output that the step size has become too small. If the integrator keeps getting stuck, it is possible that hundreds of these messages could be sent. This parameter limits the number of such messages. After this number is exceeded, the integration will still proceed; it only shuts off the publishing of the warnings.

Q.9.3.13 Maximum Number of Corrector-Iterations

max_correction_iters

- Type:unsigned int

- Default: 3

This is the number of times the corrector can be called before either:

- convergence must have been achieved, or
- the integration step fails.

Convergence tends to be achieved very quickly for orbital problems. Increasing this value is unlikely to have any noticeable effect. Note that convergence in this context indicates only that the last application of the corrector algorithm did not appreciably alter the state from the previous application of either the predictor or corrector algorithm. There are two easily confused reasons why the integrator fails to reach a stable target:

1. It fails to converge within the maximum number of iterations
2. The real solution is too far from the original predicted solution.

Once the corrector has converged, the accumulated effect of the corrector applications for this integration step is compared against the tolerance limits. If this is too high, the result is still considered unreliable and discarded. This latter test is unaffected by this parameter and it is highly unlikely that changing this parameter will 'fix' a failure indicated by the latter test.

Q.9.3.14 Maximum Usage for a Jacobian

max_num_steps_jacobian

- Type: unsigned int
- Default:20

This provides the maximum number of steps that can be taken with a given Jacobian matrix. For slowly-evolving systems with short time-steps, this number may be large. For rapidly evolving systems, it should be small. Note that this parameter is redundant if the *corrector_method* is *FunctionalIteration* because the Jacobian us not used for that process.

Q.9.3.15 Maximum Number of Convergence Failures

max_num_conv_failure

- Type:unsigned int
- Default:10

If the corrector does not converge within *max_correction_iters* iterations, it is considered failed and the integrator will back up and try again with a different (smaller) step size. If it continues to fail, doing so this many times, then the integrator itself is considered to have failed to reach the target and the integration stops.

Q.9.3.16 Maximum Number of Steps

max_num_steps

- Type:unsigned int
- Default:500

In the integration of any problem, LSODE will try to reach or exceed the target time in some number of time-steps, where the time-steps are computed internally and iterated until the integrator converges within its limits on each step. This parameter specifies the largest number of steps that the integrator can take in solving a particular problem.

For example, suppose the simulation engine called for an integration over a 60-second interval and the integrator tried steps of 1s, 2s, 2.5s, 4s, 8s, 12s, 10s, 15s, 14s, ... Summing these values indicates that the integrator has passed its first target of 60 seconds in 9 steps. If the user had stipulated that the integration had to be reached in no more than *max_num_steps = 5* steps, then the integration would have failed at 17.5s.

The step-count is incremented throughout the entire simulation, but with each new call to the integrator (a new call is one with *re_entry_point = CycleStartFinish*), the step-count at step-start is reassigned. Then the step-count for comparison against *max_num_steps* is the difference between the step-count for the simulation and the step-count at step-start.

Bibliography

[1] D. Beeman. Some Multistep Methods for Use in Molecular Dynamics Calculations. *Journal of Computational Physics*, 20:130, February 1976.

[2] Berry, M. and Healy, L. Implementation of Gauss-Jackson Integration for Orbit Propagation. *Proc. AAS/AIAA Astrodynamics Specialists Conference.*, AAS 01-426, 2001.

[3] John C Butcher. *Numerical Methods for Ordinary Differential Equations (2nd ed.).* John Wiley & Sons Inc., Hoboken, NJ, USA, 2008.

[4] Generated by doxygen. *JEOD Integration Model Reference Manual*. National Aeronautics and Space Administration, Johnson Space Center, Software, Robotics & Simulation Division, Simulation and Graphics Branch, 2101 NASA Parkway, Houston, Texas, 77058, July 2023.

[5] Hammen, D. Dynamics Manager Model. Technical Report JSC-61777-dynamics/dyn_manager, NASA, Johnson Space Center, Houston, Texas, July 2023.

[6] Hammen, D. Memory Management Model. Technical Report JSC-61777-utils/memory, NASA, Johnson Space Center, Houston, Texas, July 2023.

[7] Hammen, D. Dynamic Body Model. Technical Report JSC-61777-dynamics/dyn_body, NASA, Johnson Space Center, Houston, Texas, July 2023.

[8] Richard W Hamming. *Numerical methods for scientists and engineers (2nd ed.)*. Dover Publications, Inc., New York, NY, USA, 1986.

[9] Hindmarsh, A.C. *DLSODE Fortran code*. Center for Applied Scientific Computing, L-561, Lawrence Livermore National Laboratory, Livermore, CA 94551.

[10] Jackson, A., Thebeau, C. JSC Engineering Orbital Dynamics. Technical Report JSC-61777-docs, NASA, Johnson Space Center, Houston, Texas, July 2023.

[11] NASA. NASA Software Engineering Requirements. Technical Report NPR-7150.2, NASA, NASA Headquarters, Washington, D.C., September 2004.

[12] Radhakrishnan, K. and Hindmarsh, A.C. Description and Use of LSODE, the Livermore Solver for Ordinary Differential Equations. *NASA Reference Publication 1327.*, Lawrence Livermore national Laboratory Report UCRL-ID-113855, 1993.

[13] Lawrence Shampine. *Numerical solution of ordinary differential equations*, volume 4. Chapman and Hall, New York, NY, USA, 1994.

[14] Turner, G. Time Model. Technical Report JSC-61777-environment/time, NASA, Johnson Space Center, Houston, Texas, July 2023.