Autonomous Vehicle Simulation (AVS) Laboratory, University of Colorado

Basilisk Technical Memorandum

Document ID: Basilisk-Integrators

MODULAR DYNAMICS INTEGRATOR CAPABILITY

Prepared by	H. Schaub

Status: Initial Document
Scope/Contents
Basilisk has the capability to select a range of integration types when solve the dynamical differential equations of motion. The default integrator is a fixed time step 4th order Runge Kutta integrator. This document outlines the types of implemented integration methods, how they are validated, as well as how they can be setup.

Rev	Change Description	By	Date
1.0	Initial Revision	H. Schaub	07/2017
1.1	Small typo fixes	H. Schaub	10/18/2017
2.0	Added variable time step integrators	J. Vaz Carneiro	10/2021

Contents

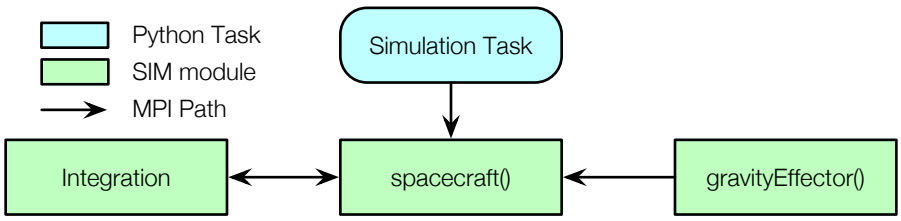

Fig. 1: Illustration of the Integrator Diagram

1 Model Description

1.1 Overview

The Basilisk integration capability is implemented in a modular manner such that different integrators can be assigned to the equations of motion that must be solved. Figure 1 illustrates how the integrator functions relative to a dynamical systems model, here the `spacecraft()` object. The ODE's are

provided by a sub-class of `DynamicObject` which must be able to respond to the `equationsOfMotion()` method. Integration of the state vector forward one time step is then handled by the `integrate` method of the integrator class. By default the `DynamicObject` is integrated using a fixed time step 4th order Runge-Kutta method.

Assume the dynamical system is given by

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}) \tag{1}$$

The initial conditions are specified through $\boldsymbol{x}_0 = \boldsymbol{x}(t_0)$. In integration time step is given through h.

1.2 Implemented Integrators

1.2.1 4th Order Runge Kutta - Default Integrator

A standard fixed time step 4th order Runge Kutta integrator is enabled by default. The 4 k_i values are defined as

$$\boldsymbol{k}_1 = \boldsymbol{f}\left(t_n, \boldsymbol{x}_n\right) \tag{2}$$

$$\boldsymbol{k}_2 = \boldsymbol{f}\left(t_n + \frac{h}{2}, \boldsymbol{x}_n + \frac{h}{2}\boldsymbol{k}_1\right) \tag{3}$$

$$\boldsymbol{k}_3 = \boldsymbol{f}\left(t_n + \frac{h}{2}, \boldsymbol{x}_n + \frac{h}{2}\boldsymbol{k}_2\right) \tag{4}$$

$$\boldsymbol{k}_4 = \boldsymbol{f}\left(t_n + h, \boldsymbol{x}_n + h\boldsymbol{k}_3\right) \tag{5}$$

The state at the next integration time $t_{n+1} = t_n + h$ is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \frac{h}{6}\left(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4\right) \tag{6}$$

1.2.2 2nd Order Runge Kutta (Heun's Method)

A 2nd order Runge-Kutta method is implemented through Heun's method.[1] The 2 k_i values are defined as

$$\boldsymbol{k}_1 = \boldsymbol{f}(t_n, \boldsymbol{x}_n) \tag{7}$$

$$\boldsymbol{k}_2 = \boldsymbol{f}(t_n + h, \boldsymbol{x}_n + h\boldsymbol{k}_1) \tag{8}$$

The state at the next integration time $t_{n+1} = t_n + h$ is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + \frac{h}{2}\left(\boldsymbol{k}_1 + \boldsymbol{k}_2\right) \tag{9}$$

1.2.3 1st Order Runge Kutta (Euler's Method)

A first order Runge-Kutta method is implemented through Euler's method. The one k_1 value is defined as

$$\boldsymbol{k}_1 = \boldsymbol{f}(t_n, \boldsymbol{x}_n) \tag{10}$$

The state at the next integration time $t_{n+1} = t_n + h$ is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h\boldsymbol{k}_1 \tag{11}$$

[1] `http://goo.gl/SWdyBZ`

1.2.4 4th Order Runge Kutta Fehlberg Variable Time Step

A fourth order Runge-Kutta-Fehlberg is implemented. It propagates the state using a fourth order integration method, while comparing the resulting truncation error with a fifth order integration method. The k_i values are defined as

$$\boldsymbol{k}_1 = \boldsymbol{f}\left(t_n, \boldsymbol{x}_n\right) \tag{12}$$

$$\boldsymbol{k}_2 = \boldsymbol{f}\left(t_n + \frac{h}{4}, \boldsymbol{x}_n + \frac{h}{4}\boldsymbol{k}_1\right) \tag{13}$$

$$\boldsymbol{k}_3 = \boldsymbol{f}\left(t_n + \frac{3h}{8}, \boldsymbol{x}_n + \frac{3h}{32}\boldsymbol{k}_1 + \frac{9h}{32}\boldsymbol{k}_2\right) \tag{14}$$

$$\boldsymbol{k}_4 = \boldsymbol{f}\left(t_n + \frac{12h}{13}, \boldsymbol{x}_n + \frac{1932h}{2197}\boldsymbol{k}_1 - \frac{7200h}{2197}\boldsymbol{k}_2 + \frac{7296h}{2197}\boldsymbol{k}_3\right) \tag{15}$$

$$\boldsymbol{k}_5 = \boldsymbol{f}\left(t_n + h, \boldsymbol{x}_n + \frac{439h}{216}\boldsymbol{k}_1 - 8h\boldsymbol{k}_2 + \frac{3680h}{513}\boldsymbol{k}_3 - \frac{8450h}{4104}\boldsymbol{k}_4\right) \tag{16}$$

$$\boldsymbol{k}_6 = \boldsymbol{f}\left(t_n + \frac{h}{2}, \boldsymbol{x}_n - \frac{8h}{27}\boldsymbol{k}_1 + 2h\boldsymbol{k}_2 - \frac{3544h}{2565}\boldsymbol{k}_3 + \frac{1859h}{4104}\boldsymbol{k}_4 - \frac{11h}{40}\boldsymbol{k}_5\right) \tag{17}$$

The state at the next integration time $t_{n+1} = t_n + h$ is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h\left(\frac{25}{216}\boldsymbol{k}_1 + \frac{1408}{2565}\boldsymbol{k}_3 + \frac{2197}{4104}\boldsymbol{k}_4 - \frac{1}{5}\boldsymbol{k}_5\right) \tag{18}$$

The estimate for the relative truncation error is

$$\boldsymbol{\delta} = h\frac{\left\|\frac{1}{360}\boldsymbol{k}_1 - \frac{128}{4275}\boldsymbol{k}_3 - \frac{2197}{75240}\boldsymbol{k}_4 - \frac{1}{50}\boldsymbol{k}_5 + \frac{2}{55}\boldsymbol{k}_6\right\|}{\|\boldsymbol{x}_n\|} \tag{19}$$

The updated time step is calculated through the following equation

$$h_{\text{new}} = 0.9h\left(\frac{\boldsymbol{\epsilon}}{\boldsymbol{\delta}}\right)^{1/5}, \tag{20}$$

where $\boldsymbol{\epsilon}$ corresponds to the relative tolerance and its default value is 10^{-4}. If the norm of the state is smaller than the absolute tolerance (default value of 10^{-8}), the relative error is calculated with respect to that absolute tolerance instead of the true vector norm. The time step is scaled by 0.9 for robustness, so that the integrator will not use values that barely pass through the relative tolerance.
The algorithm for the variable time step integrator works as follows:

1. For every state vector, compute the $\boldsymbol{k}_i$ integration weights through equations 12-17.
2. Propagate the state to the next time step using those weights using equation 18.
3. For every state, calculate the relative truncation error (19) and store the largest value of all the state errors.
4. Compute the new time step through 20.
5. If the relative truncation error $\boldsymbol{\delta}$ is larger than the relative tolerance $\boldsymbol{\epsilon}$, repeat steps 1-4 with the new time step until it does.
6. Update the integration time with the time step used during integration: $t_{n+1} = t_n + h$.

7. Update the integration time step with the new time step: $h = h_{\text{new}}$.
8. Check if the new time step would overpass the final integration time. If it does, change it so that $h = t_f - t_{n+1}$.
9. Go back to step 1 until the final integration time has been reached.

1.2.5 7th Order Runge Kutta Fehlberg Variable Time Step

A seventh order Runge-Kutta-Fehlberg is implemented. It propagates the state using a seventh order integration method, while comparing the resulting truncation error with an eighth order integration method. The k_i values are defined in a general way as

$$\boldsymbol{k}_1 = \boldsymbol{f}\left(t_n + \boldsymbol{\alpha}_1 h, \boldsymbol{x}_n\right) \tag{21}$$

$$\boldsymbol{k}_i = \boldsymbol{f}\left(t_n + \boldsymbol{\alpha}_i h, \boldsymbol{x}_n + h\sum_{j=1}^{i-1}\boldsymbol{\beta}_{ij}\boldsymbol{k}_j\right), \quad i = 2, ..., 13 \tag{22}$$

where the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ matrices are defines as

$$\boldsymbol{\alpha} = \begin{bmatrix} 0 \\ \frac{2}{27} \\ \frac{1}{9} \\ \frac{1}{6} \\ \frac{5}{12} \\ \frac{1}{2} \\ \frac{5}{6} \\ \frac{1}{6} \\ \frac{2}{3} \\ \frac{1}{3} \\ 1 \\ 0 \\ 1 \end{bmatrix} \quad \boldsymbol{\beta} = \begin{bmatrix} & & & & & & & & & & & \\ \frac{2}{27} & & & & & & & & & & & \\ \frac{1}{36} & \frac{1}{12} & & & & & & & & & & \\ \frac{1}{24} & 0 & \frac{1}{8} & & & & & & & & & \\ \frac{5}{12} & 0 & -\frac{25}{16} & -\frac{25}{16} & & & & & & & & \\ \frac{1}{20} & 0 & 0 & \frac{1}{4} & \frac{1}{5} & & & & & & & \\ -\frac{25}{108} & 0 & 0 & \frac{125}{108} & -\frac{65}{27} & \frac{125}{54} & & & & & & \\ \frac{31}{300} & 0 & 0 & 0 & \frac{61}{225} & -\frac{2}{9} & \frac{13}{900} & & & & & \\ 2 & 0 & 0 & -\frac{53}{6} & \frac{704}{45} & -\frac{107}{9} & \frac{67}{90} & 3 & & & & \\ -\frac{91}{108} & 0 & 0 & \frac{23}{108} & -\frac{976}{135} & \frac{311}{54} & -\frac{19}{60} & \frac{17}{6} & -\frac{1}{12} & & & \\ \frac{2383}{4100} & 0 & 0 & -\frac{341}{164} & \frac{4496}{1025} & -\frac{301}{82} & \frac{2133}{4100} & \frac{45}{82} & \frac{45}{164} & \frac{18}{41} & & \\ \frac{3}{205} & 0 & 0 & 0 & 0 & -\frac{6}{41} & -\frac{3}{205} & -\frac{3}{41} & \frac{3}{41} & \frac{6}{41} & 0 & \\ -\frac{1777}{4100} & 0 & 0 & -\frac{341}{164} & \frac{4496}{1025} & -\frac{289}{82} & \frac{2193}{4100} & \frac{51}{82} & \frac{33}{164} & \frac{12}{41} & 0 & 1 \end{bmatrix} \tag{23}$$

The state at the next integration time $t_{n+1} = t_n + h$ is

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n + h\sum_{i=1}^{13}\mathsf{CH}(i)\boldsymbol{k}_i \tag{24}$$

and the truncation error is

$$\boldsymbol{\delta} = h\frac{\left\|\sum_{i=1}^{13}\mathsf{CT}(i)\boldsymbol{k}_i\right\|}{\|\boldsymbol{x}_n\|} \tag{25}$$

The CH and CT matrices are given by

$$\mathsf{CH} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{34}{105} \\ \frac{9}{35} \\ \frac{9}{35} \\ \frac{9}{280} \\ \frac{9}{280} \\ 0 \\ \frac{41}{840} \\ \frac{41}{840} \end{bmatrix} \qquad \mathsf{CT} = \begin{bmatrix} -\frac{41}{840} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -\frac{41}{840} \\ \frac{41}{840} \\ \frac{41}{840} \end{bmatrix} \tag{26}$$

The algorithm used to update the time step is the same as the one described for the 4th order variable time step integrator.

2 Model Functions

The Basilisk integrator functionality is not a regular BSK module, but rather works in conjunction with the `DynamicalObject` class. The integration functions and goals are:

- **Default Integrator**: The dynamical object should have a default integrator assigned when created. This avoids the user having to add an integrator in Python, unless they want to select an alternate integrator.
- **Works on any equations of motion**: The integrator needs to function on any dynamical system.

3 Model Assumptions and Limitations

3.1 Assumptions

The equations of motion class `DynamicalObject` is assumed to respond to the method `equationsOfMotion`. The integrator then integrates the system forward in time one BSK time step.

3.2 Limitations

4 Test Description and Success Criteria

As the integrator functionality is not a regular BSK module with specified input/output behavior, the integration can only be tested in an integrated test. The integrated test employed is located in:

`src/tests/scenarios/test_scenarioIntegrators.py`

4.1 Test inputs

Each simulation uses the point-mass spacecraft equations of motion

$$\ddot{\boldsymbol{r}} = -\frac{\mu}{r^3}\boldsymbol{r} \tag{27}$$

with the initial orbit elements shown in Table 2. The only gravitational body considered is Earth. The simulation time for each case is 3/4 of an orbit period. Each implemented Integrator method is tested using the above initial conditions and a large time step of 120 seconds. This large time step makes the integrator errors more easily visible between the integration methods.

Table 2: Initial Spacecraft Ephemeris

Element	Description	Value
a	Semi-Major Axis	7000km
e	Eccentricity	0.0001
i	Inclination Angle	33.3°
Ω	Ascending Node	33.3°
ω	Argument of Periapses	48.2°
f	True Anomaly	347.8°

4.2 Test sections

The same simulation setup is run for each of the integration methods:

1. The first test uses the default RK4 integration method. Here the simulation script does not specify the integration method, testing both that the RK4 integrator is the default integration method, and that that the integrator is implemented correctly.
2. The 2nd test uses the RKF45 integration method.
3. The 3rd test used the RK78 integration method.
4. The 4th test uses the Euler's integration method.
5. the 5th test uses Heun's integration method.

The resulting data points are illustrated in Figure 2. The large 120 second time step causes all the integration methods to significantly deviate from the truth locations (shown in black). The integrated validation test ensures that the BSK integrations yield the same integration corruptions to validate the mathematical implementation.

4.3 Test success criteria

The integrated position states are checked at 5 even data points along the simulation time again pre-computed truth answers. These truth answers were generated in Mathematica implementing the same initial conditions, and replicating the integration math. The accuracy threshold is set to 1 meter, a small value compared to the 7,000,000 meter near-circular orbit radius.

5 Test Parameters

Three tests are run controlled through a single test parameter called `integratorCase`. The possible values are shown in Table 3.

6 Test Results

All integration checks within the integrated test `scenarios/test_scenarioIntegrators.py` passed. Table 4 shows the test results, while Figure 2 shows the resulting trajectories for each integration test.

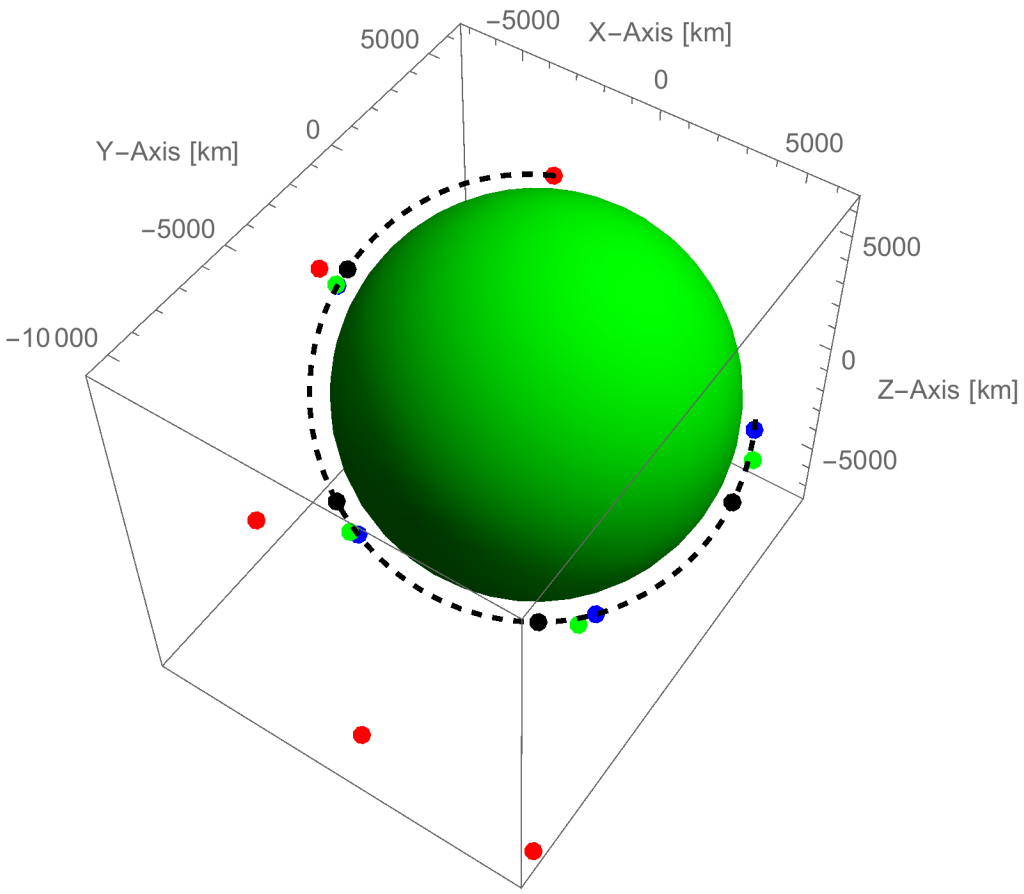

Fig. 2: Illustration of the true orbit position samples (Black) versus the RK4 (Blue), RK2 (Green) and RK1 (Red) integration results.

7 User Guide

7.1 Not Specifying an Integration Method

If a Python BSK simulation is setup without specifying an integration method, then the default behavior is to load the fixed time step 4th order Runge-Kutta (RK4) method. No additional code is required by the user.

7.2 Selecting Alternate Integration Methods

Assume the `DynamicObject` class is the `spacecraft()` object, declared through

```
scObject = spacecraft.spacecraft()
```

To invoke the Euler's integration scheme, the corresponding integration module is created using

Table 3: Error tolerance for each test.

Test	Tolerated Error
"rk4"	1 meter
"rkf45"	1 meter
"rkf78"	1 meter
"euler"	1 meter
"rk2"	1 meter

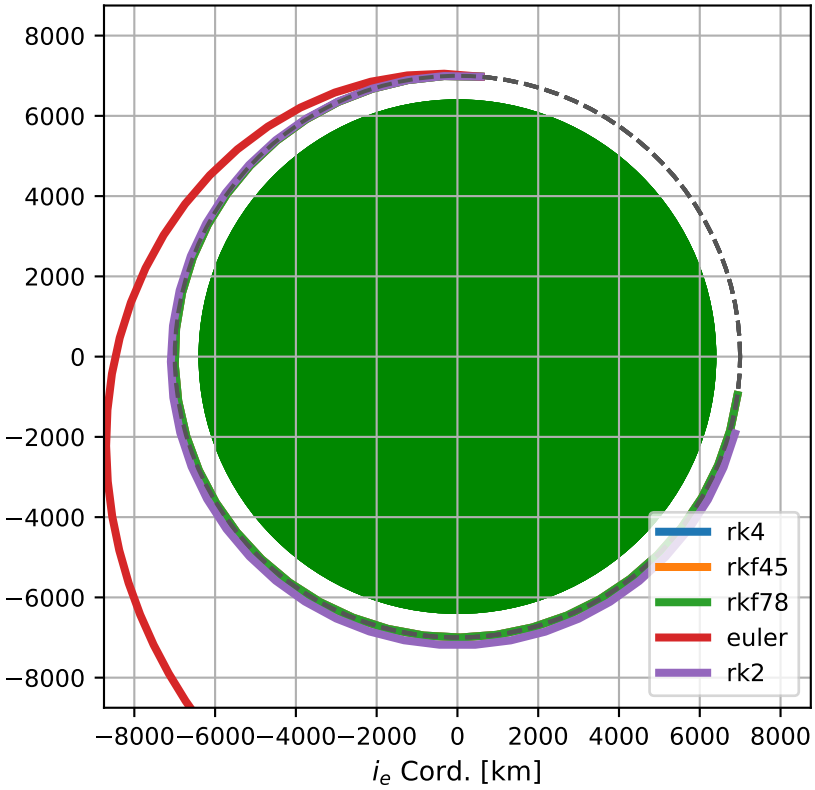

Fig. 3: Illustration of the BSK integrated trajectories

Table 4: Integration test results.

Test	Pass/Fail	BSK Error Notes
"rk4"	PASSED	
"rkf45"	PASSED	
"rkf78"	PASSED	
"euler"	PASSED	
"rk2"	PASSED	

```
integratorObject = svIntegrators.svIntegratorEuler(scObject)
```

If the 2nd order Heun's integration method is desired, use instead

```
integratorObject = svIntegrators.svIntegratorRK2(scObject)
```

To force the default RK4 method, use

```
integratorObject = svIntegrators.svIntegratorRK4(scObject)
```

If the 4th order variable time step integration method is desired, use instead

```
integratorObject = svIntegrators.svIntegratorRKF45(scObject)
```

If the 7th order variable time step integration method is desired, use instead

```
integratorObject = svIntegrators.svIntegratorRKF78(scObject)
```

Next, to connect this integrator module to the `DynamicObject` instance (i.e. `spacecraft()` called `scObject`, use the following code

```
scObject.setIntegrator(integratorObject)
```

That is it, the Basilisk simulation is now setup to use the desired numerical integration method. If the user is using a variable time step and wants to override the default tolerance values, add the following lines of code

```
integratorObject.absTol = 1e-6
integratorObject.relTol = 1e-3
```

7.3 Creating New Integration Methods

New integration modules can readily be created for Basilisk. They are all stored in the folder

```
Basilisk/simulation/dynamics/Integrators/
```

The integrators must be created to function on a general state vector and be independent of the particular dynamics being integrated. Note that the default integrator is placed inside the `_GeneralModulesFiles` folder within the dynamics folder.