spacejam Documentation

Release 1.0.0

Ian Weaver, Sherif Gerges, Lauren Yoo

Dec 12, 2018

Contents

for automatic differentiation and other looney things

CHAPTER 1

Introduction

Numerical integration is a powerful tool that can be used to simulate the evolution of a wide range of systems. Given the initial conditions of your particular system, `spacejam` numerically integrates your system using pre-baked implicit schemes that are powered by automatic differentiation.

In this documentation, we outline the technique of automatic differentiation and numerical integration, go through the implicit schemes included in our library, and show some neat applications to astronomy and ecology.

CHAPTER 2

How to use

The following series of demos will step through how to differentiate a wide variety of functions with `spacejam`. Check out *Installation* to get started.

2.1 Demo I: Scalar function, scalar input

This is the simplest case, where the function you provide takes in a single scalar argument $(x = a)$ and outputs a single scalar value $f(a)$.

For example, let's take a look at the function $f(x) = x^3$, which you can define below as:

```
import numpy as np

def f(x):
    return np.array([x**3])
```

All `spacejam` needs now is for you to specify a point $\mathbf{p}$ where you would like to evaluate your function at:

```
p = np.array([5])  # evaluation point
```

Now, evaluating your function and simultaneously computing the derivative with `spacejam` at this point is as easy as:

```
import spacejam as sj

ad = sj.AutoDiff(f, p)
```

The real part of `ad` is now $f(x = 5) = 125$ and the dual part is $\frac{\mathrm{d}f}{\mathrm{d}x}\Big|_{x=5} = 75$.

These real and dual parts are conveniently stored, respectively, as the `r` and `d` attributes in `ad` and can easily be printed to examine:

```
print(f'f(x) evaluated at p:\n{ad.r}\n\n'
      f'derivative of f(x) evaluated at p:\n{ad.d}')
```

```
f(x) evaluated at p:
[125.00]

derivative of f(x) evaluated at p:
[75.00]
```

Note: `numpy` arrays are used when defining your function and returning results because `spacejam` can also operate on multivariable functions and parameters, which we outline in *Demo II: Scalar function with vector input*. and *Demo III: Vector function with vector input*.

2.2 Demo II: Scalar function with vector input

This next demo explores the case where a new example function f can accept vector input, for example $\mathbf{p} = (x_1, x_2) = (5, 2)$ and return a single scalar value $f(\mathbf{p}) = f(x_1, x_2) = 3x_1x_2 - 2x_2^3/x_1$

The dual number objects are created in much the same way as in *Demo I*, where:

$$\begin{aligned} p_{x_1} &= f(x_1, x_2) + \epsilon_{x_1}\frac{\partial f}{\partial x_1} - \epsilon_{x_2}0 \\ p_{x_2} &= f(x_1, x_2) + \epsilon_{x_1}0 - \epsilon_{x_2}\frac{\partial f}{\partial x_2} \end{aligned} \quad ,$$

as described in *Automatic Differentiation: A brief overview*. Internally, this is accomplished with the `idx` and `x` argument in `spacejam.dual` so that it knows which dual parts need to be set to zero in the modified dual numbers above. `spacejam.autodiff` then performs the following internally:

$$f(\mathbf{p}) + \epsilon_{x_1}\frac{\partial f}{\partial x_1} - \epsilon_{x_2}\frac{\partial f}{\partial x_2} \equiv f(\mathbf{p}) + \epsilon\left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right] = f(\mathbf{p}) + \epsilon\nabla f \quad .$$

tl;dr: all that needs to be done is:

```
import numpy as np
import spacejam as sj

def f(x_1, x_2):
    return np.array([3*x_1*x_2 - 2*x_2**3/x_1])

p = np.array([5, 2]) # evaluation point (x_1, x_2) = (5, 2)

ad = sj.AutoDiff(f, p) # create spacejam object

# check out the results
print(f'f(x) evaluated p:\n{ad.r}\n\n'
      f'grad of f(x) evaluated at p:\n{ad.d}')
```

```
f(x) evaluated p:
[26.80]

grad of f(x) evaluated at p:
[6.64 10.20]
```

2.3 Demo III: Vector function with vector input

This final demo shows how to use `spacejam` to simultaneously evaluate the example vector function:

$$\mathbf{F} = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \\ f_3(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_1^2 + x_1 x_2 + 2 \\ x_1 x_2^3 + x_1^2 \\ x_2^3/x_1 + x_1 + x_1^2 x_2^2 + x_2^4 \end{bmatrix}$$

and its Jacobian:

$$\mathbf{J} = \begin{bmatrix} \nabla f_1(x_1, x_2) \\ \nabla f_2(x_1, x_2) \\ \nabla f_3(x_1, x_2) \end{bmatrix} \quad .$$

at the point $\mathbf{p} = (x_1, x_2) = (1, 2)$.

The interface with `spacejam` happens to be exactly the same as in the previous two demos, only now your $F(x)$ will return a 1D `numpy` array of functions (f_1, f_2, f_3):

```
# your (m) system of equations:
# F(x_1, x_2, ..., x_m) = (f1, f2, ..., f_n)
def F(x_1, x_2):
        f_1 = x_1**2 + x_1*x_2 + 2
        f_2 = x_1*x_2**3 + x_1**2
        f_3 = x_1 + x_1**2*x_2**2 + x_2**3/x_1 + x_2**4
        return np.array([f_1, f_2, f_3])

# where you want them evaluated at:
# p = (x_1, x_2, ..., x_m)
p = np.array([1, 2])

# auto differentiate!
ad = sj.AutoDiff(F, p)

# check out the results
print(f'F(x) evaluated at p:\n{ad.r}\n\n'
      f'Jacobian of F(x) evaluated at p:\n{ad.d}')
```

```
F(x) evaluated at p:
[[5.00]
 [9.00]
 [29.00]]

Jacobian of F(x) evaluated at p:
[[4.00 1.00]
 [10.00 12.00]
 [1.00 48.00]]
```

Internally, for each i th entry, in the 1D `numpy` array `ad._full`, the real part is the i th component of $\mathbf{F}(\mathbf{p})$ and the dual part is the corresponding row in the Jacobian $\mathbf{J}$ evaluated at $\mathbf{p} = (x_1, x_2) = (1, 2)$.

This is done in *`spacejam.autodiff.AutoDiff._matrix`* for you with:

```
Fs = np.empty((F(*p).size, 1)) # initialze empty F(p)
jac = np.empty((F(*p).size, p.size)) # initialize empty J F(p)

for i, f in enumerate(ad._full): # fill in each row of each
    Fs[i] = f.r
```

(continues on next page)

(continued from previous page)

```
    jac[i] = f.d

print(f'formated F(p):\n{Fs}\n\nformated J F(p):\n{jac}')
```

```
formated F(p):
[[5.00]
 [9.00]
 [29.00]]

formated J F(p):
[[4.00 1.00]
 [10.00 12.00]
 [1.00 48.00]]
```

where `ad._full` looks like:

```
print(ad._full)
```

```
[5.00 + eps [4.00 1.00] 9.00 + eps [10.00 12.00] 29.00 + eps [1.00 48.00]]
```

Note: You are also free to make your own dual numbers (for example $z = 3 + \epsilon\ 4$) by doing:

```
z = sj.Dual(3, 4)

print(z)
```

```
3.00 + eps 4.00
```

We also use `numpy` to overload basic trig functions, exponential, and natural log, which are not builtins in python. This is accessed by doing:

```
result = np.cos(z)
print(result)
```

```
-0.99 - eps 0.56
```

`spacejam` formats all numbers to two decimal places but internally the whole number is stored.

CHAPTER 3

Background

3.1 Numerical Integration: A brief crash couse

Many physical systems can be expressed as a series of differential equations. Euler's method is the simplest numerical procedure for solving these equations given some initial conditions. In the case of our problem statement:

- We have some initial conditions (such as position and velocity of a planet) and we want to know what kind of orbit this planet will trace out, given that only the force acting on it is gravity.
- Using the physical insight that the "slope" of position over time is velocity, and the "slope" of velocity over time is acceleration, we can predict, or integrate, how the quantities will change over time.
- More explicitly, we can use the acceleration supplied by gravity to predict the velocity of our planet, and then use this velocity to predict its position a timestep Δt later.
- This gives us a new position and the whole process starts over again at the next timestep. Here is a schematic of the Euler integration method.

This plot above could represent the component of the planet's velocity varies over time. Specifically, we have some solution curve (black) that we want to approximate (red), given that we only know two things:

- where we started (t_0, y_0)
- the rate of how where we were changes with time $\left(\dot{y}_0 \equiv \frac{\mathrm{d}y_0}{\mathrm{d}t} = \frac{y_1 - y_0}{h}\right)$

The cool thing about this is that even though we do not explicity know what y_1 is, the fact that we are given $\dot{y}_0$ from the initial conditions allows us to bootstrap our way around this. Starting with the definition of slope, we can use the timestep $h \equiv \Delta t = t_{n+1} - t_n$, to find where we will be a timestep later $\dot{y}_1$:

$$\dot{y}_0 = \frac{y_1 - y_0}{h} \quad \longrightarrow \quad y_1 = y_0 + h\dot{y}_0 \quad .$$

Generalizing to any timestep n:

$$y_{n+1} = y_n + h\dot{y}_n \quad .$$

Whenever all of the $n+1$ terms are on one side of the equation and the n terms are on the other, we have an **explicit numerical method**. This can also be extended to k components for y_n with the simple substitution:

$$y_n \longrightarrow \underline{\mathrm{X}}_n = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{pmatrix}, \quad \dot{y}_n \longrightarrow \underline{\dot{\mathrm{X}}}_n = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_k \end{pmatrix}, \quad y_{n+1} \longrightarrow \underline{\mathrm{X}}_{n+1} = \underline{\mathrm{X}}_n + h\underline{\dot{\mathrm{X}}}_n \quad .$$

This is intuitively straightforward and easy to implement, but there is a downside: the solutions **do not converge** for any given timestep. If the steps are too large, our numerical estimations are essentially dominated by progation of error and would return results that are non-physical, and if they are too small the simulation would take too long to run.

We need a scheme that remains stable and accurate for a wide range of timesteps, which is what **implicit differentiation** can accomplish. An example of one such scheme is:

$$\underline{\mathrm{X}}_{n+1} = \underline{\mathrm{X}}_n + h\underline{\dot{\mathrm{X}}}_{n+1} \quad .$$

Now we have $n+1$ terms on both sides, making this an implicit scheme. This is know as the backward Euler method and a common way of solving this and many other similar schemes that build of off this one is by re-casting it as a root finding problem. For the backward Euler method, this would look like:

$$\underline{\mathrm{g}}(\underline{\mathrm{X}}_{n+1}) = \underline{\mathrm{X}}_{n+1} - \underline{\mathrm{X}}_n - h\underline{\dot{\mathrm{X}}}_{n+1} \quad .$$

Here, the root of the new function $\underline{\mathrm{g}}$ is the solution to our original implicit integration equation. The Newton-Raphson method is a useful root finding algorithm, but one of its steps requires the computation of the $k \times k$ Jacobian:

$$\underline{\mathrm{J}}(\underline{\mathrm{X}}_{n+1}) = \nabla_{n+1}\underline{\mathrm{X}}_{n+1} = \begin{pmatrix} \nabla(\dot{x}_1)_{n+1} \\ \nabla(\dot{x}_2)_{n+1} \\ \vdots \\ \nabla(\dot{x}_k)_{n+1} \end{pmatrix} \quad .$$

Note: We avoid using superscript notation here because that will be reserved for identifying iterates in Newton's method, which we discuss in *Example Applications*.

`spacejam` can also support systems with a different number of equations than variables, i.e. non-square Jacobians. See *Demo III: Vector function with vector input*.

Accurately computing the elements of the Jacobian can be numerically expensive, so a method to quickly and accurately compute derivatives would be extremely useful. `spacejam` provides this capability by computing the Jacobian quickly and accurately via *automatic differentiation*, which can be used to solve a wide class of problems that depend on implicit differentiation for numerically stable solutions.

We walk through using `spacejam` to implement Newton's method for the Backward Euler method and its slightly more sophisticated siblings, the $s = 1$ and $s = 2$ Adams-Moulton methods in *Example Applications*. Note: $s = 0$ is just the original backward Euler method and $s = 1$ is also know as the famous trapezoid rule. To the best of our knowledge, there is not a cool name for the $s = 2$ method.

3.2 Automatic Differentiation: A brief overview

This is a method to simultaneously compute a function and its derivative to machine precision. This can be done by introducing the dual number $\epsilon^2 = 0$, where $\epsilon \neq 0$. If we transform some arbitrary function $f(x)$ to $f(x+\epsilon)$ and expand it, we have:

$$f(x+\epsilon) = f(x) + \epsilon f'(x) + O(\epsilon^2) \quad .$$

By the definition of ϵ, all second order and higher terms in ϵ vanish and we are left with $f(x+\epsilon) = f(x) + \epsilon f'(x)$, where the dual part, $f'(x)$, of this transformed function is the derivative of our original function $f(x)$. If we adhere to the new system of math introduced by dual numbers, we are able to compute derivatives of functions exactly.

For example, multiplying two dual numbers $z_1 = a_r + \epsilon a_d$ and $z_2 = b_r + \epsilon b_d$ would behave like:

$$\begin{aligned} z_1 \times z_2 &= (a_r + \epsilon a_d) \times (b_r + \epsilon b_d) = a_r b_r + \epsilon(a_r b_d + a_d b_r) + \epsilon^2 a_d b_d \\ &= \boxed{a_r b_r + \epsilon(a_r b_d + a_d b_r)} \quad . \end{aligned}$$

A function like $f(x) = x^2$ could then be automatically differentiated to give:

$$f(x) \longrightarrow f(x+\epsilon) = (x+\epsilon) \times (x+\epsilon) = x^2 + \epsilon(x \cdot 1 + 1 \cdot x) = x^2 + \epsilon\, 2x \quad ,$$

where $f(x) + \epsilon f'(x)$ is returned as expected. Operations like this can be redefined via **operator overloading**, which we implement in *Implementation Details*. This method is also easily extended to multivariable functions with the introduction of "dual number basis vectors" $\underline{\mathrm{p}}_i = i + \epsilon_i 1$, where i takes on any of the components of $\underline{\mathrm{X}}_n$. For example, the multivariable function $f(x, y) = xy$ would transform like:

$$\begin{aligned} cancel x \quad &\longrightarrow \quad \underline{\mathrm{p}}_x = x + \epsilon_x + \epsilon_y\, 0 \\ y \quad &\longrightarrow \quad \underline{\mathrm{p}}_y = y + \epsilon_x\, 0 + \epsilon_y \\ f(x,y) \quad &\longrightarrow \quad f(\underline{\mathrm{p}}) = (x + \epsilon_x + \epsilon_y\, 0) \times (y + \epsilon_x\, 0 + \epsilon_y) \\ &\qquad\quad = xy + \epsilon_y x + \epsilon_x y + \epsilon_x \epsilon_y \\ &\qquad\quad = xy + \epsilon_x y + \epsilon_y x \quad , \end{aligned}$$

where we now have:

$$\begin{aligned} f(x+\epsilon_x, y+\epsilon_y) &= f(x,y) + \epsilon_x fx + \epsilon_y fy = f(x,y) + \epsilon\,[fx,\ fy] \\ &= f(x,y) + \epsilon \nabla f(x,y) \quad . \end{aligned}$$

This is accomplished internally in `spacejam.autodiff.Autodiff._ad` with:

```
    def _ad(self, func, p, kwargs=None):
        """ Internally computes `func(p)` and its derivative(s).

        Notes
        -----
        `_ad` returns a nested 1D `numpy.ndarray` to be formatted internally
        accordingly in :any:`spacejam.autodiff.AutoDiff.__init__` .

        Parameters
        ----------
        func : numpy.ndarray
            function(s) specified by user.
        p : numpy.ndarray
            point(s) specified by user.
        """
        if len(p) == 1: # scalar p
            p_mult = np.array([dual.Dual(p)])

        else:# vector p
            p_mult = [dual.Dual(pi, idx=i, x=p) for i, pi in enumerate(p)]
            p_mult = np.array(p_mult) # convert list to numpy array

        # perform AD with specified function(s)
        if kwargs:
```

(continues on next page)

(continued from previous page)

```
            result = func(*p_mult, **kwargs)
        else:
            result = func(*p_mult)
        return result
```

The `x` argument in the `spacejam.dual.Dual` class above sets the length of the p dual basis vector and the `idx` argument sets the proper index to 1 (with the rest being zero).

CHAPTER 4

Software Organization

Cool tree cartoon of main files:

```
spacejam
├── LICENSE.txt
├── MANIFEST.in¶
├── README.md
├── requirements.txt
├── setup.cfg
├── setup.py
└── spacejam
    ├── __init__.py
    ├── autodiff.py
    ├── dual.py
    ├── integrators.py
    └── test
        ├── __init__.py
        ├── test_autodiff.py
        ├── test_dual.py
        └── test_integrators.py
```

4.1 Overview of main modules

- *spacejam.autodiff*: Performs automatic differentiation of user-specified functions by following dual number rules provided by *spacejam.dual*
- *spacejam.dual*: Overloads basic math operations and returns an automatic differentiation `spacejam` object
- *spacejam.integrators*: Suite of implicit integration schemes

4.2 Installation

4.2.1 Virtual Environment

For development, or just to have a self contained enviroment to use `spacejam` in, run the following commands anywhere on your computer:

```
python -m venv venv
source venv/bin/activate
pip install spacejam
```

Optional: If you prefer working in a Jupyter notebook, you can also

4.2.2 Tests

Unit tests are stored in `spacejam/tests` and each module mentioned above has its own doctests. TravisCI and Coveralls integration is also provided. You can run these tests and coverage reports yourself by doing the following:

```
cd venv/lib/python3.7/site-packages/spacejam
pytest --doctest-modules --cov=. --cov-report term-missing
```

Check out *How to use* for a quickstart tutorial.

CHAPTER 5

Implementation Details

5.1 Data Structures

- `spacejam` uses 1D `numpy.ndarrays` to return partial derivatives, where the j th entry contains $\frac{\partial f_i}{\partial x_j}$ for $i = 1, ...m$ and $j = 1, ...k$. In general, this is for m different functions that are a function of k different variables.
- The internal convenience function *spacejam.autodiff.AutoDiff._matrix* stacks these 1D arrays into an $m \times k$ `numpy.ndarray` Jacobian matrix for ease of viewing, as described in *Demo III: Vector function with vector input*.

5.2 API

5.2.1 spacejam.autodiff

class spacejam.autodiff.**AutoDiff**(*func*, *p*, *kwargs=None*)

Performs automatic differentiation (AD) on functions input by user.

AD if performed by transforming *f(x1, x2, . . .)* to *f(p_x1, p_x2, . . .)*, where *p_xi* is returned from *spacejam.dual.Dual* . The final result is then returned in a series of 1D *numpy.ndarray* or formatted matrices depending on if the user specified functions F are multivariable or not.

r

numpy.ndarray – User defined function(s) *F* evaluated at *p*.

d

numpy.ndarray – Corresponding derivative, gradient, or Jacobian of user defined functions(s).

__init__(*func*, *p*, *kwargs=None*)

Parameters

- **func** (*numpy.ndarray*) – user defined function(s).

- **p** (*numpy.ndarray*) – user defined point(s) to evaluate derivative/gradient/Jacobian at.

_ad (*func*, *p*, *kwargs=None*)
Internally computes *func(p)* and its derivative(s).

Notes

_ad returns a nested 1D *numpy.ndarray* to be formatted internally accordingly in *spacejam.autodiff.AutoDiff.__init__* .

Parameters

- **func** (*numpy.ndarray*) – function(s) specified by user.
- **p** (*numpy.ndarray*) – point(s) specified by user.

_matrix (*F*, *p*, *result*)
Internally formats *result* returned by *spacejam.autodiff.AutoDiff._ad* into matrices.

Parameters

- **F** (*numpy.ndarray*) – functionss specified by user.
- **p** (*numpy.ndarray*) – point(s) specified by user.
- **result** (*numpy.ndarray*) – Nested 1D numpy.ndarray to be formatted into matrices.

Returns

- **Fs** (*numpy.ndarray*) – Column matrix of functions evaluated at points(s).
- **jac** (*numpy.ndarray*) – Corresponding Jacobian matrix.

5.2.2 spacejam.dual

class spacejam.dual.**Dual** (*real*, *dual=None*, *idx=None*, *x=array(1)*)
Creates dual numbers and defines dual number math.

A real number *a* is taken in and its dual counterpart *a + eps [1.00]* is returned to facilitate automatic differentiation in *spacejam.autodiff* .

Notes

The dual part can optionally be returned as a "dual basis vector" [0 1 0] if the user function f is multivariable and the partial derivative $\partial f/\partial x_2$ is desired, for example.

r
float – real part of *spacejam.dual.Dual* .

d
numpy.ndarray – dual part of *spacejam.dual.Dual* .

__add__ (*other*)
Returns the addition of self and other

Parameters

- **self** (*Dual object*) –
- **other** (*Dual object, float, or int*) –

Returns z

Return type Dual object that is the sum of self and other

Examples

```
>>> z = Dual(1, 2) + Dual(3, 4)
>>> print(z)
4.00 + eps 6.00
>>> z = 2 + Dual(1, 2)
>>> print(z)
3.00 + eps 2.00
```

__init__(*real*, *dual=None*, *idx=None*, *x=array(1)*)

Parameters

- **real** (*int/float*) – real part of *spacejam.dual.Dual* .
- **dual** (*float*) – dual part of *spacejam.dual.Dual* (default 1.00) .
- **idx** (*int (default None)*) – index in dual part of dual basis vector.
- **x** (*numpy.ndarray (default [1])*) – set size of pre-allocated array for dual basis vector.

__mul__(*other*)

Returns the product of self and other

Parameters

- **self** (*Dual object*) –
- **other** (*Dual object, float, or int*) –

Returns z

Return type Dual object that is the product of self and other

Examples

```
>>> z = Dual(1, 2) * Dual(3, 4)
>>> print(z)
3.00 + eps 10.00
>>> z = 2 * Dual(1, 2)
>>> print(z)
2.00 + eps 4.00
```

__neg__()

Returns negation of self

Examples

```
>>> z = Dual(1, 2)
>>> print(-z)
-1.00 - eps 2.00
```

__pos__()
Returns self

Examples

```
>>> z = Dual(1, 2)
>>> print(+z)
1.00 + eps 2.00
```

__pow__(*other*)
Performs (self.r + eps self.d) ** (other.r + eps other.d)

Parameters

- **self** (*Dual object*) –
- **other** (*Dual object, float, or int*) –

Returns z

Return type Dual object that is self raised to the other power

Examples

```
>>> z = Dual(1, 2) ** Dual(3, 4)
>>> print(z)
1.00 + eps 6.00
```

__radd__(*other*)
Returns the addition of self and other

Parameters

- **self** (*Dual object*) –
- **other** (*Dual object, float, or int*) –

Returns z

Return type Dual object that is the sum of self and other

Examples

```
>>> z = Dual(1, 2) + Dual(3, 4)
>>> print(z)
4.00 + eps 6.00
>>> z = 2 + Dual(1, 2)
>>> print(z)
3.00 + eps 2.00
```

__repr__()
Prints self in the form a_r + eps a_d, where self = Dual(a_r, a_d), a_r and a_d are the real and dual part of self, respectively, and both terms are automatically rounded to two decimal places

Returns z

Return type Dual object that is the product of self and other

Examples

```
>>> z = Dual(1, 2)
>>> print(z)
1.00 + eps 2.00
```

__rmul__(*other*)
Returns the product of self and other

Parameters

- **self** (*Dual object*) –
- **other** (*Dual object, float, or int*) –

Returns z

Return type Dual object that is the product of self and other

Examples

```
>>> z = Dual(1, 2) * Dual(3, 4)
>>> print(z)
3.00 + eps 10.00
>>> z = 2 * Dual(1, 2)
>>> print(z)
2.00 + eps 4.00
```

__rsub__(*other*)
Returns the subtraction of other from self

Parameters

- **self** (*Dual object*) –
- **other** (*Dual object, float, or int*) –

Returns z – difference of other and self

Return type Dual object

Examples

```
>>> z = 2 - Dual(1, 2)
>>> print(z)
1.00 - eps 2.00
```

__rtruediv__(*other*)
Returns the quotient of other and self

Parameters

- **self** (*Dual object*) –
- **other** (*Dual object, float, or int*) –

Returns z

Return type Dual object that is the product of self and other

Examples

```
>>> z = 2 / Dual(1, 2)
>>> print(z)
2.00 - eps 4.00
```

__sub__(*other*)
Returns the subtraction of self and other

Parameters

- **self** (*Dual object*) –
- **other** (*Dual object, float, or int*) –

Returns **z** – difference of self and other

Return type Dual object

Notes

Subtraction does not commute in general. A specialized __rsub__ is required.

Examples

```
>>> z = Dual(1, 2) - Dual(3, 4)
>>> print(z)
-2.00 - eps 2.00
>>> z = Dual(1, 2) - 2
>>> print(z)
-1.00 + eps 2.00
```

__truediv__(*other*)
Returns the quotient of self and other

Parameters

- **self** (*Dual object*) –
- **other** (*Dual object, float, or int*) –

Returns **z**

Return type Dual object that is the quotient of self and other

Examples

```
>>> z = Dual(1, 2) / 2
>>> print(z)
0.50 + eps 1.00
>>> z = Dual(3, 4) / Dual(1, 2)
>>> print(z)
3.00 - eps 2.00
```

cos()
Returns the cosine of a

Parameters **self** (*Dual object*) –

Returns z

Return type cosine of self

Examples

```
>>> z = np.cos(Dual(0, 1))
>>> print(z)
1.00 + eps -0.00
```

exp()
Returns e**self

Parameters **self** (*Dual object*) –

Returns z

Return type e**self

Examples

```
>>> z = np.exp(Dual(1, 2))
>>> print(z)
2.72 + eps 5.44
```

sin()
Returns the sine of a

Parameters **self** (*Dual object*) –

Returns z

Return type sine of self

Examples

```
>>> z = np.sin(Dual(0, 1))
>>> print(z)
0.00 + eps 1.00
```

tan()
Returns the tangent of a

Parameters **self** (*Dual object*) –

Returns z

Return type tangent of self

Examples

```
>>> z = np.tan(Dual(0,1))
>>> print(z)
0.00 + eps 1.00
```

5.2.3 spacejam.integrators

spacejam.integrators.**amsi** (*func*, *X_old*, *h=0.001*, *X_tol=0.1*, *i_tol=100.0*, *kwargs=None*)
First order Adams-Moulton method (AKA Trapezoid)

Parameters

- **func** (*function*) – User defined function to be integrated.
- **X_old** (*numpy.ndarray*) – Initial input to user function
- **h** (*float (default 1E-3)*) – Timestep
- **X_tol** (*float (default 1E-1)*) – Minimum difference between Newton-Raphson iterates to terminate on.
- **i_tol** (*int (default 1E2)*) – Maximum number of Newton-Raphson iterations. Entire simulation terminates if this number is exceeded.
- **kwargs** (*dict (default None)*) – optional arguments to be supplied to user defined function.

Returns **X_new** – Final X_n+1 found from root finding of implicit method

Return type numpy.ndarray

spacejam.integrators.**amsii** (*func*, *X_n*, *X_nn*, *h=0.001*, *X_tol=0.1*, *i_tol=100.0*, *kwargs=None*)
Second order Adams-Moulton method

Parameters

- **func** (*function*) – User defined function to be integrated.
- **X_n** (*numpy.ndarray*) – X_n
- **X_nn** (*numpy.ndarray*) – X_n-1
- **h** (*float (default 1E-3)*) – Timestep
- **X_tol** (*float (default 1E-1)*) – Minimum difference between Newton-Raphson iterates to terminate on.
- **i_tol** (*int (default 1E2)*) – Maximum number of Newton-Raphson iterations. Entire simulation terminates if this number is exceeded.
- **kwargs** (*dict (default None)*) – optional arguments to be supplied to user defined function.

Returns **X_new** – Final X_n+1 found from root finding of implicit method

Return type numpy.ndarray

spacejam.integrators.**amso** (*func*, *X_old*, *h=0.001*, *X_tol=0.1*, *i_tol=100.0*, *kwargs=None*)
Zeroth order Adams-Moulton method (AKA Backward Euler)

Parameters

- **func** (*function*) – User defined function to be integrated.
- **X_old** (*numpy.ndarray*) – Initial input to user function

- **h** (*float (default 1E-3)*) – Timestep
- **X_tol** (*float (default 1E-1)*) – Minimum difference between Newton-Raphson iterates to terminate on.
- **i_tol** (*int (default 1E2)*) – Maximum number of Newton-Raphson iterations. Entire simulation terminates if this number is exceeded.
- **kwargs** (*dict (default None)*) – optional arguments to be supplied to user defined function.

Returns **X_new** – Final X_n+1 found from root finding of implicit method

Return type numpy.ndarray

Examples

```
>>>
```

CHAPTER 6

Example Applications

6.1 Background

`spacejam` can be used to simulate a wide range of physical systems. To accomplish this, we provide an integration suite of implicit solvers that draw from the first three orders of the Adams-Moulton methods. These methods can be accessed from *`spacejam.integrators`* and each use the root finding Newton-Raphson method with an initial forward Euler guess. We will now describe each implicit scheme and how to go about using it with `spacejam`.

6.1.1 (s = 0) Method

As we saw in *Numerical Integration: A brief crash couse*, this is a numerical scheme to solve the implicit equation:

$$\underline{\mathrm{X}}_{n+1} = \underline{\mathrm{X}}_n + h\dot{\underline{\mathrm{X}}}_{n+1}$$

by re-casting it as the root finding problem:

$$\underline{\mathrm{g}}(\underline{\mathrm{X}}_{n+1}) = \underline{\mathrm{X}}_{n+1} - \underline{\mathrm{X}}_n - h\dot{\underline{\mathrm{X}}}_{n+1} \quad .$$

In 1D, the Newton-Raphson method successively finds better and better approximations to the root of a function $f(x)$ in the following way:

- Make a guess next to one of the roots you want
- Draw the corresponding tangent line (red) at this guess evaluated on the original function (blue)
- Trace this tangent line to where it intercepts the x-axis
- Make this intercept your new guess
- Rinse and repeat until your latest guess is close enough (your tolerance) to the root you wanted to approximate in the first place

The equation for this can be quickly derived by solving for the next root iterate x_{n+1} from the definition of the derivative:

$$\text{slope} = \frac{\text{rise}}{\text{run}} \quad \longrightarrow \quad f'(x_n) = \frac{f(x_n)}{x_n - x_{n+1}} \quad \longrightarrow \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad .$$

This is naturally extended to vector functions that accept multi-valued input by using the multi-variable version of the derivative, the Jacobian $\underline{\mathrm{J}}$:

$$\underline{\mathrm{X}}_{n+1} = \underline{\mathrm{X}}_n - \underline{\mathrm{J}}[\underline{\mathrm{f}}(\underline{\mathrm{X}}_n)]^{-1}\underline{\mathrm{f}}(\underline{\mathrm{X}}_n) \quad ,$$

where:

$$\begin{aligned} \underline{\mathrm{J}}[\underline{\mathrm{f}}(\underline{\mathrm{X}}_n)]_{ij} &= f_i x_j \quad , \\ \underline{\mathrm{f}}\,(\underline{\mathrm{X}}_n) &= [f_1, f_2, \cdots, f_m] \quad , \\ \underline{\mathrm{X}}_n &= [x_1, x_2, \cdots, x_k] \quad . \end{aligned}$$

For these examples, $m = k$, $\underline{\mathrm{f}} = \underline{\dot{x}}_n = [\dot{x}_1, \dot{x}_2, \cdots, \dot{x}_m]_n$, $1 \leq i, j \leq k$.

Applying this to our backward Euler equation:

$$\begin{aligned} 0 = \underline{\mathrm{g}}\left(\underline{\mathrm{X}}_{n+1}\right) &= \underline{\mathrm{X}}_{n+1} - \underline{\mathrm{X}}_n - h\dot{\underline{\mathrm{X}}}_{n+1} \quad , \\ \underline{\mathrm{X}}_{n+1}^{(i+1)} &= \underline{\mathrm{X}}_{n+1}^{(i)} - \underline{\mathrm{D}}\left[\underline{\mathrm{g}}\left(\underline{\mathrm{X}}_{n+1}\right)^{(i)}\right]^{-1} \underline{\mathrm{g}}\left(\underline{\mathrm{X}}_{n+1}\right)^{(i)} \quad . \end{aligned}$$

Here, (i) and $(i+1)$ have been used to avoid confusion with the n and $n+1$ iterate used in the 1D example above, and the root to this equation is the solution $\underline{\mathrm{X}}_{n+1}$ to our original implicit equation. The Jacobian $\underline{\mathrm{J}}$ is hiding inside of $\underline{\mathrm{D}}$ and we can make it show itself by just performing the multi-variable derivative that is required of the Newton-Raphson method:

$$cancel\underline{\mathrm{D}}\left[\underline{\mathrm{g}}\left(\underline{\mathrm{X}}_{n+1}\right)^{(i)}\right] = \underline{\mathrm{g}}\left(\underline{\mathrm{X}}_{n+1}\right)^{(i)}\underline{\mathrm{X}}_{n+1}^{(i)} = \underline{\mathrm{X}}_{n+1}^{(i)}\underline{\mathrm{X}}_{n+1}^{(i)} - 0\underline{\mathrm{X}}_n^{(i)}\underline{\mathrm{X}}_{n+1}^{(i)} - h\underline{\dot{x}}_{n+1}\underline{\mathrm{X}}_{n+1}^{(i)} = \underline{\mathrm{I}} - h\underline{\mathrm{J}}\left[\left(\underline{\dot{x}}_{n+1}\right)^{(i)}\right] \quad ,$$

where $\underline{\mathrm{I}}$ is the identity matrix. All that is needed now is an initial guess for $\underline{\mathrm{X}}_{n+1}^{(0)}$ to jump start Newton's method. A single forward Euler step should do:

$$\begin{aligned} \underline{\mathrm{X}}_{n+1}^{(0)} &= \underline{\mathrm{X}}_n^{(0)} + h\underline{\dot{x}}_n^{(0)} \quad , \\ \underline{\dot{x}}_n^{(0)} &= \begin{bmatrix} \dot{x}_1(t=0) \\ \dot{x}_2(t=0) \\ \vdots \\ \dot{x}_k(t=0) \end{bmatrix} \quad . \end{aligned}$$

In this framework, both the real and dual part of the dual object returned by `spacejam` will be used. To summarize:

- The user supplies the system of equations $\underline{\dot{x}}_n$ and initial conditions $\underline{\dot{x}}_n^{(0)}$.
- The user implements the integration scheme using the real part returned from `spacejam` for $\underline{\dot{x}}_n^{(i)}$ and the dual part as $\underline{\mathrm{J}}\left[\left(\underline{\dot{x}}_{n+1}\right)^{(i)}\right]$.

6.1.2 (s = 1) Method

A similar implementation can be made with the next order up in this family of implicit methods. In this scheme we have:

$$\underline{\mathrm{X}}_{n+1} = \underline{\mathrm{X}}_n + \frac{1}{2}h\left(\underline{\dot{x}}_{n+1} + \underline{\dot{x}}_n\right) \quad .$$

Applying the same treatment of turning this into a root finding problem and applying Newton's method gives the similar result:

$$\begin{aligned}
\underline{\mathrm{g}}(\underline{\mathrm{X}}_{n+1}) &= \underline{\mathrm{X}}_{n+1} - \underline{\mathrm{X}}_n - \frac{h}{2}\underline{n}_{n+1} - \frac{h}{2}\underline{n} \quad , \\
\underline{\mathrm{X}}_{n+1}^{(i+1)} &= \underline{\mathrm{X}}_{n+1}^{(i)} - \underline{\mathrm{D}}\left[\underline{\mathrm{g}}\left(\underline{\mathrm{X}}_{n+1}\right)^{(i)}\right]^{-1} \underline{\mathrm{g}}\left(\underline{\mathrm{X}}_{n+1}\right)^{(i)} \quad , \\
\underline{\mathrm{D}} &= \underline{\mathrm{I}} - \frac{h}{2}\underline{\mathrm{J}}\left[\left(\underline{n}_{n+1}\right)^{(i)}\right] \quad .
\end{aligned}$$

In this new scheme, $\underline{\mathrm{D}}$ has an extra factor of $1/2$ on its Jacobian in the backward and now `spacejam` will also be computing $\underline{n}$.

6.1.3 (s = 2) Method

In this final scheme we have:

$$\underline{\mathrm{X}}_{n+1} = \underline{\mathrm{X}}_n + h\left(\frac{5}{12}\underline{n}_{n+1} + \frac{2}{3}\underline{n} - \frac{1}{12}\underline{n}_{n-1}\right) \quad .$$

The corresponding $\underline{\mathrm{g}}$ and $\underline{\mathrm{D}}$ are then:

$$\begin{aligned}
\underline{\mathrm{g}} &= \underline{\mathrm{X}}_{n+1} - \underline{\mathrm{X}}_n - h\left(\frac{5}{12}\underline{n}_{n+1} + \frac{2}{3}\underline{n} - \frac{1}{12}\underline{n}_{n-1}\right) \quad , \\
\underline{\mathrm{D}} &= \underline{\mathrm{I}} - \frac{5h}{12}\underline{\mathrm{J}}\left[\left(\underline{n}_{n+1}\right)^{(i)}\right] \quad .
\end{aligned}$$

Note: Each of the three methods above are implemented in *spacejam.integrators*. The tolerance determining when to end Newton-Raphson iterations and the break point in number of iterations can also respectively be controlled by the keyword arguments `X_tol` and `i_tol` in all integrator functions.

We demonstrate each method in our example systems below.

6.2 Astronomy Example

6.2.1 Background

In this example, we will integrate the orbits of a hypothetical three-body star-planet-moon system. This exercise is motivated by the first potential discovery of an exomoon made not too long ago.

In 2D Cartesian coordinates, the equations of motion that govern the orbit of body A due to bodies B and C are:

- $\dot{x}_A = v_{x_A}$
- $\dot{y}_A = v_{y_A}$
- $\dot{v}_{x_A} = \dfrac{Gm_B}{d_{AB}^3}(x_B - x_A) + \dfrac{Gm_C}{d_{AC}}(x_C - x_A)$
- $\dot{v}_{y_A} = \dfrac{Gm_B}{d_{AB}^3}(y_B - y_A) + \dfrac{Gm_C}{d_{AC}}(y_C - y_A) \quad ,$

where the following definitions are given:

- (x_i, y_i): positional coordinates of body i, with mass m_i

- (v_{x_i}, v_{y_i}): components of body i's velocity
- d_{ij}: distance between body i and body j
- G: Universal Gravitational Constant (as far as we know)

We will be using an external package (astropy) that is not included in `spacejam` for this demonstration. This step is totally optional, but it makes using units and physical constants a lot more convenient.

6.2.2 Initial Conditions

For this toy model, let's place a 10 Jupiter mass exoplanet 0.01 AU to the left of a sun-like star, which we place at the origin. Let's also have this exoplanet orbit this star with the typical Keplerian velocity $v = \sqrt{GM/r}$, starting in the negative y direction, where M is the mass of the star and *r* is the distance of this exoplanet from its star.

Next, let's place an exomoon with $1/1000$ th the mass of the exoplanet about $110,000$ km to the left of this exoplanet. This ensures that the exomoon is within its gravitational sphere of influence. Let's also have this exomoon start moving with Keplerian speed in the negative y direction. note: this would be the sum of the exoplanet's velocity and the Keplerian speed of the moon due to just the gravitational influence of the exoplanet.

Finally, let's pick a time step that goes something like a tenth of the time it would initially take the exomoon to fall straight into the planet if it didn't happen to have any Keplerian speed. To a certain extent, this choice is pretty arbitrary because of implicit schemes' relative insensitivity to time step size relative to those for explicit schemes, but our implicit solving implementation does partially rely on an explicit scheme, so it's still important to consider.

```
import numpy as np
from astropy import units as u
from astropy import constants as c

# constants
solMass   = (1 * u.solMass).cgs.value
solRad    = (1 * u.solRad).cgs.value
jupMass   = (1 * u.jupiterMass).cgs.value
jupRad    = (1 * u.jupiterRad).cgs.value
earthMass = (1 * u.earthMass).cgs.value
earthRad  = (1 * u.earthRad).cgs.value
G         = (1 * c.G).cgs.value
AU        = (1 * u.au).cgs.value
year      = (1 * u.year).cgs.value
day       = (1 * u.day).cgs.value
earth_v   = (30 * u.km/u.s).value
moon_v    = (1 * u.km/u.s).cgs.value

# mass ratio of companion to secondary
q             = 0.001
# primary
host_mass     = solMass
host_rad      = solRad
# secondary
scndry_mass   = 10*jupMass
scndry_rad    = 1.7*jupRad
scndry_x      = -0.01*AU
scndry_y      = 0.0
scndry_vx     = 0.0
scndry_vy     = -np.sqrt(G*host_mass/np.abs(scndry_x)) # assuming Keplerian for now
# companion
cmpn_mass     = q*scndry_mass
cmpn_rad      = 0.3*scndry_rad
```

(continues on next page)

(continued from previous page)

```
hill_sphere    = np.abs(scndry_x) * (scndry_mass / (3*host_mass))**(1/3)
cmpn_x         = scndry_x - 0.5 * hill_sphere
cmpn_y         = scndry_y
cmpn_vx        = 0.0
cmpn_vy        = scndry_vy - np.sqrt(G*scndry_mass/(0.5 * hill_sphere))

m_1 = host_mass # host star
m_2 = scndry_mass #m_1 / 5000 # hot jupiter
m_3 = cmpn_mass  # companion

# m1: primary (hardcoded)
x_1  =  0.0
y_1  =  0.0
vx_1 =  0.0
vy_1 =  0.0

# m2: secondary
x_2  = scndry_x
y_2  = scndry_y # doesn't matter where it starts on y because of symmetry of system
vx_2 = scndry_vx
vy_2 = scndry_vy # assuming Keplerian for now

# m3: companion
x_3  = cmpn_x
y_3  = cmpn_y
vx_3 = cmpn_vx
vy_3 = cmpn_vy

# characteristic timescale set by secondary's orbital timescale
T0 = 2*np.pi*np.sqrt(np.abs(scndry_x)**3/(G*m_1))
tmax  = 2.5*T0

uold_1 = np.array([x_1, y_1, vx_1, vy_1])
uold_2 = np.array([x_2, y_2, vx_2, vy_2])
uold_3 = np.array([x_3, y_3, vx_3, vy_3])

m1_coord = uold_1
m2_coord = uold_2
m3_coord = uold_3

r0 = np.sqrt( (uold_3[0] - uold_2[0])**2 + (uold_3[1] - uold_2[1])**2 )
v0 = np.sqrt(uold_3[2]**2 + uold_3[3]**2)
f = -1
h = 10**(f) * r0 / v0
N = 1500 # number of steps to run sim

# Store initial positions and velocities
uold_1 = np.array([x_1, y_1, vx_1, vy_1]) # star
uold_2 = np.array([x_2, y_2, vx_2, vy_2]) # exoplanet
uold_3 = np.array([x_3, y_3, vx_3, vy_3]) # exomoon
```

6.2.3 Equations of Motion

The system of differential equations governing our system look like:

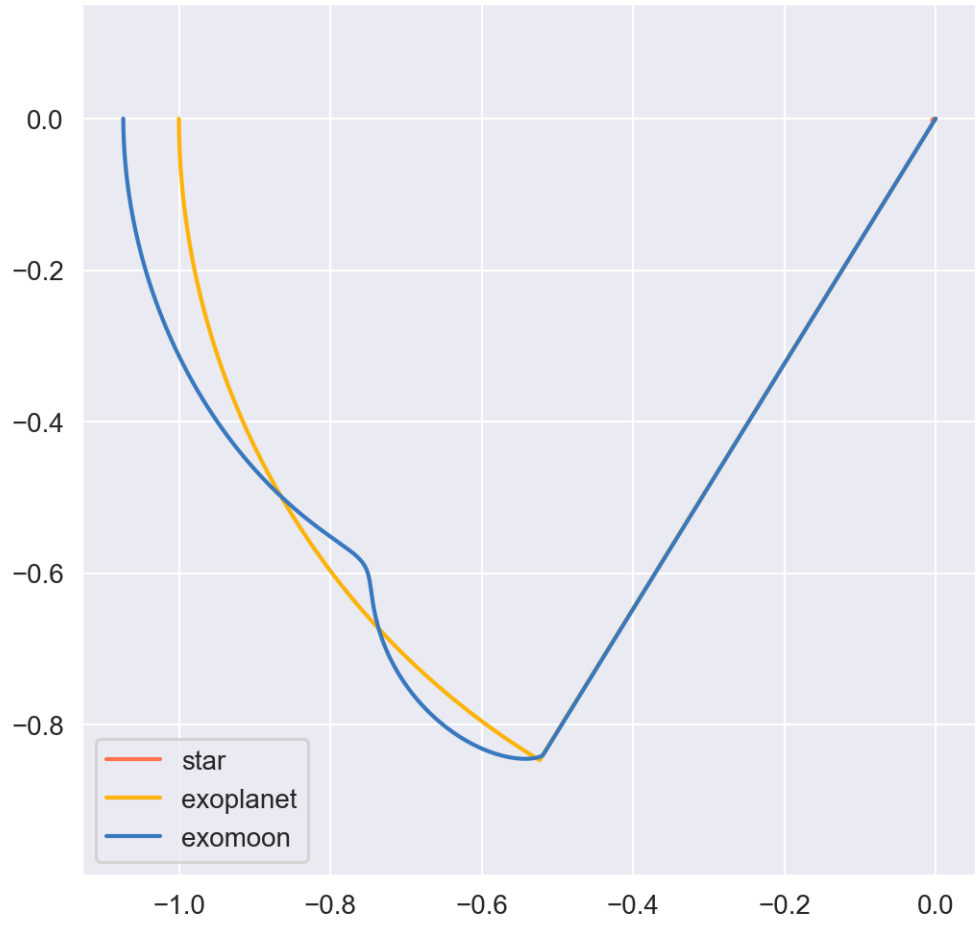
```
def f(x, y, vx, vy, uold_b=None, mb=0, uold_c=None, mc=0):
    # position and velocity
    r_a = np.array([x, y])
    v_a = np.array([vx, vy])

    r_b = uold_b[:2]
    r_c = uold_c[:2]

    # position vector pointing from one of the two masses to m_i
    d_ab = np.linalg.norm(r_b - r_a)
    d_ac = np.linalg.norm(r_c - r_a)

    # calulating accelerations
    gx = G*mb/d_ab**3 * (r_b[0] - x) + (G*mc/d_ac**3) * (r_c[0] - x)
    gy = G*mb/d_ab**3 * (r_b[1] - y) + (G*mc/d_ac**3) * (r_c[1] - y)

    # return derivatives
    f1 = vx
    f2 = vy
    f3 = gx
    f4 = gy
    return np.array([f1, f2, f3, f4])
```

6.2.4 Simulation

Our toy model can now be run with `spacejam` and its included suite of integrators to produce the following orbits.

(s = 0)

```
import spacejam as sj

X_1 = np.zeros((N, uold_1.size))
X_1[0] = uold_1
X_2 = np.zeros((N, uold_2.size))
X_2[0] = uold_2
X_3 = np.zeros((N, uold_3.size))
X_3[0] = uold_3

for n in range(N-1):
    kwargs_1 = {'uold_b': X_2[n], 'mb': m_2, 'uold_c': X_3[n], 'mc': m_3}
    X_1[n+1] = sj.integrators.amso(f, X_1[n], h=h, kwargs=kwargs_1)

    kwargs_2 = {'uold_b': X_1[n], 'mb': m_1, 'uold_c': X_3[n], 'mc': m_3}
    X_2[n+1] = sj.integrators.amso(f, X_2[n], h=h, kwargs=kwargs_2)

    kwargs_3 = {'uold_b': X_1[n], 'mb': m_1, 'uold_c': X_2[n], 'mc': m_2}
    X_3[n+1] = sj.integrators.amso(f, X_3[n], h=h, kwargs=kwargs_3)

    # stop iterating if Newton-Raphson method does not converge
    if X_1[n+1] is None or X_2[n+1] is None or X_3[n+1] is None:
        break
```



Note: Axes are scaled by the initial distance of the exoplanet from its host star and oriented in the usual XY fashion.

This integration scheme actually fails partway through the simulation. `spacejam` provides the following suggestions to fix this in its error message:

```
SystemExit:
Sorry, spacejam did not converge for s=0 A-M method.
Try adjusting X_tol, i_tol, or using another integrator.
```

We will follow the last suggestion and use the higher order s=1 scheme instead.

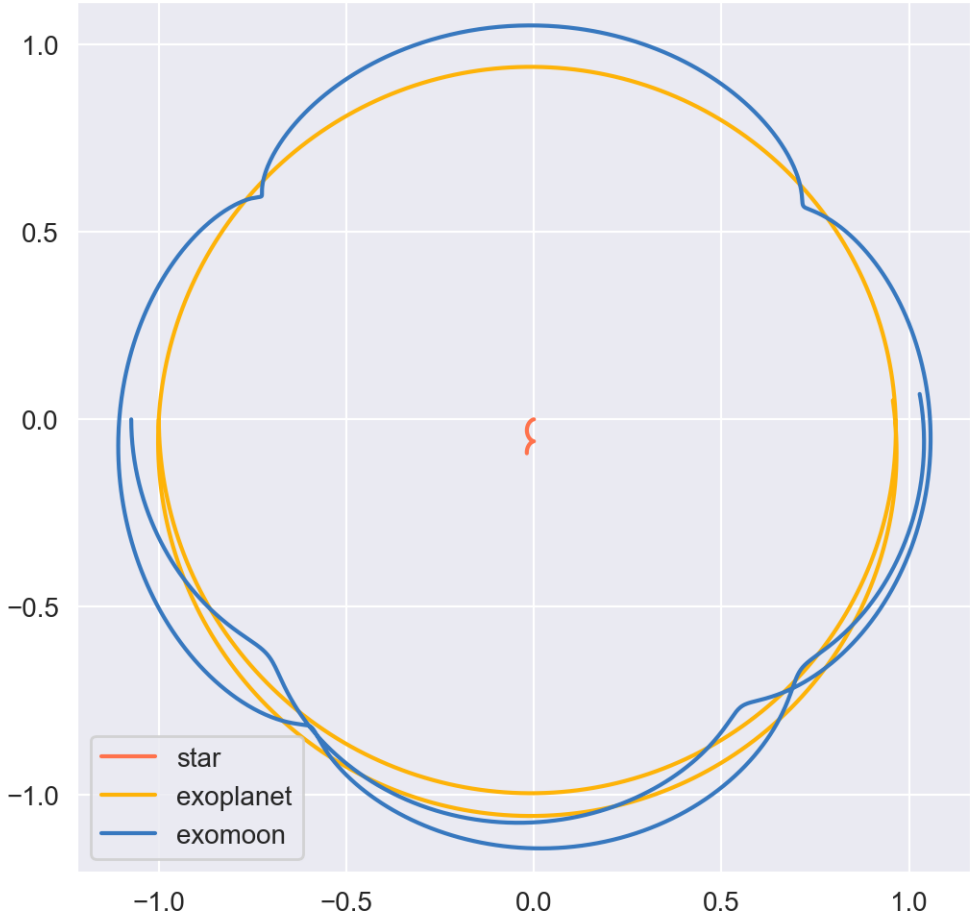
(s = 1)

```
X_1 = np.zeros((N, uold_1.size))
X_1[0] = uold_1
X_2 = np.zeros((N, uold_2.size))
X_2[0] = uold_2
X_3 = np.zeros((N, uold_3.size))
X_3[0] = uold_3

for n in range(N-1):
```

(continues on next page)

(continued from previous page)

```
    kwargs_1 = {'uold_b': X_2[n], 'mb': m_2, 'uold_c': X_3[n], 'mc': m_3}
    X_1[n+1] = sj.integrators.amsi(f, X_1[n], h=h, kwargs=kwargs_1)

    kwargs_2 = {'uold_b': X_1[n], 'mb': m_1, 'uold_c': X_3[n], 'mc': m_3}
    X_2[n+1] = sj.integrators.amsi(f, X_2[n], h=h, kwargs=kwargs_2)

    kwargs_3 = {'uold_b': X_1[n], 'mb': m_1, 'uold_c': X_2[n], 'mc': m_2}
    X_3[n+1] = sj.integrators.amsi(f, X_3[n], h=h, kwargs=kwargs_3)

    # stop iterating if Newton-Raphson method does not converge
    if X_1[n+1] is None or X_2[n+1] is None or X_3[n+1] is None:
        break
```



It works! Let's go up another order.

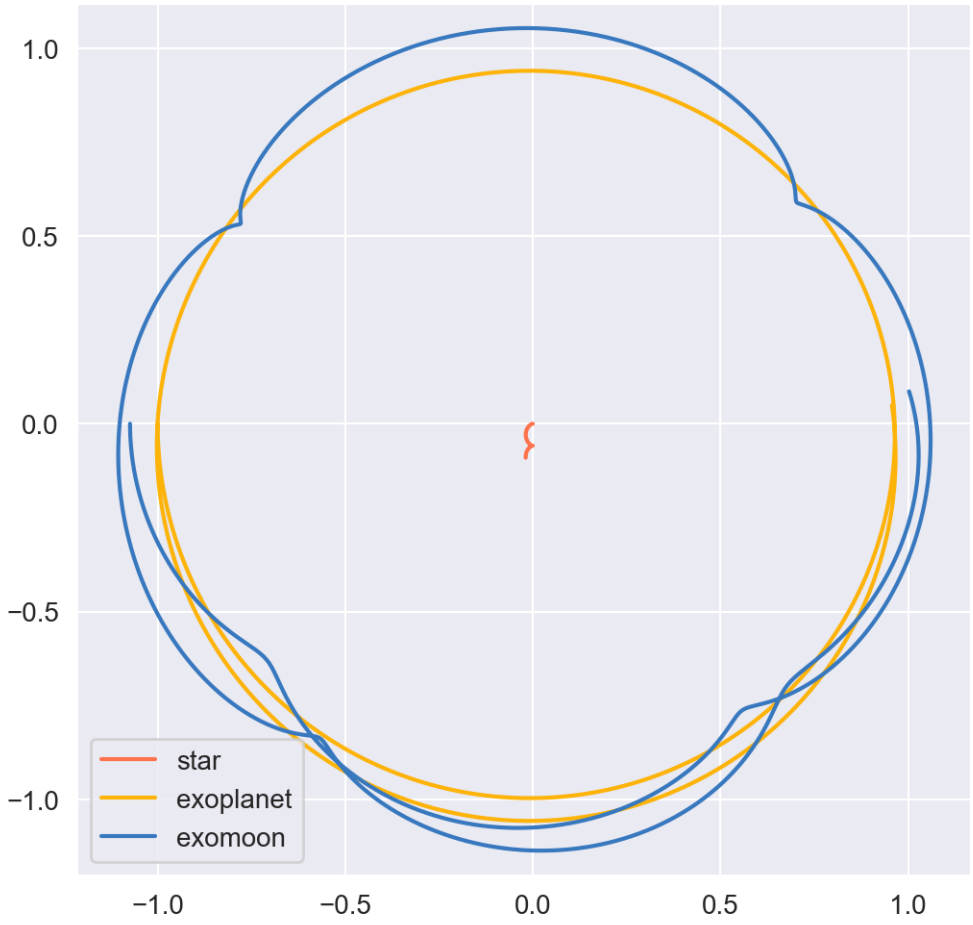
(s = 2)

```
X_1 = np.zeros((N, uold_1.size))
X_1[0] = uold_1
X_2 = np.zeros((N, uold_2.size))
X_2[0] = uold_2
X_3 = np.zeros((N, uold_3.size))
```

(continues on next page)

(continued from previous page)

```
X_3[0] = uold_3

# This method requires the 2nd step as well to get started. Will
# just use a forward Euler guess for this
kwargs_1 = {'uold_b': X_2[0], 'mb': m_2, 'uold_c': X_3[0], 'mc': m_3}
ad = sj.AutoDiff(f, X_1[0], kwargs=kwargs_1)
X_1[1] = X_1[0] + h*ad.r.flatten()

kwargs_2 = {'uold_b': X_1[0], 'mb': m_1, 'uold_c': X_3[0], 'mc': m_3}
ad = sj.AutoDiff(f, X_2[0], kwargs=kwargs_2)
X_2[1] = X_2[0] + h*ad.r.flatten()

kwargs_3 = {'uold_b': X_1[0], 'mb': m_1, 'uold_c': X_2[0], 'mc': m_2}
ad = sj.AutoDiff(f, X_3[0], kwargs=kwargs_3)
X_3[1] = X_3[0] + h*ad.r.flatten()

for n in range(1, N-1):
    kwargs_1 = {'uold_b': X_2[n], 'mb': m_2, 'uold_c': X_3[n], 'mc': m_3}
    X_1[n+1] = sj.integrators.amsii(f, X_1[n], X_1[n-1],
                                    h=h, kwargs=kwargs_1)

    kwargs_2 = {'uold_b': X_1[n], 'mb': m_1, 'uold_c': X_3[n], 'mc': m_3}
    X_2[n+1] = sj.integrators.amsii(f, X_2[n], X_2[n-1],
                                    h=h, kwargs=kwargs_2)

    kwargs_3 = {'uold_b': X_1[n], 'mb': m_1, 'uold_c': X_2[n], 'mc': m_2}
    X_3[n+1] = sj.integrators.amsii(f, X_3[n], X_3[n-1],
                                    h=h, kwargs=kwargs_3)

    # stop iterating if Newton-Raphson method does not converge
    if X_1[n+1] is None or X_2[n+1] is None or X_3[n+1] is None:
        break
```



Note: All plots for this example were styled with the external package seaborn and created with the following snippet below:

```
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style('darkgrid')

fig, ax = plt.subplots(figsize=(6, 6))
ax.set_aspect('equal', 'datalim')

# normalize plot axes
a_0 = np.linalg.norm(m2_coord)

# custom colors
c1 = sns.xkcd_palette(["pinkish orange"])[0]
c2 = sns.xkcd_palette(["amber"])[0]
c3 = sns.xkcd_palette(["windows blue"])[0]

ax.plot(X_1[:,0]/a_0, X_1[:,1]/a_0, c=c1, label='star')
ax.plot(X_2[:,0]/a_0, X_2[:,1]/a_0, c=c2, label='exoplanet')
ax.plot(X_3[:,0]/a_0, X_3[:,1]/a_0, c=c3, label='exomoon')

ax.legend()
```

Static images can be a bit difficult to interpret, so we also included a stylized movie for the final plot.

Note: Everything is still scaled by the initial distance of the exoplanet from its star.

An analysis of the change in total energy and angular momentum of the system each step in the simulation would be a good diagnostic to see which integration scheme is actually giving the most accurate results.

Now we turn to a completely different example that can also be handled with `spacejam`.

6.3 Ecology Example

6.3.1 Background

In this example, we look at a popular system of differential equations used to describe the dynamics of biological systems where two sets of species (predator and prey) interact. The population of each can be tracked with:

$$xt = \dot{x} = \alpha x - \beta xy \quad ,$$
$$yt = \dot{y} = \delta xy - \gamma y \quad ,$$

where,

- x: number of prey
- y: number of predators
- $\dot{x}$ and $\dot{y}$: instantaneous growth rate of the prey and predator populations, respectively
- $\alpha, \beta, \delta, \gamma$: parameters describing interactions of the two species

6.3.2 Initial Conditions

We will test this system with the initial conditions that are known to produce a stable system.

```
import numpy as np

N = 1000
h = .01 # timestep
X_0 = np.array([2., 1.]) # initial population conditions ([prey, predator])
X = np.zeros((N, X_0.size))
X[0] = X_0
```

6.3.3 Equations of population growth

The system can be created with the following:

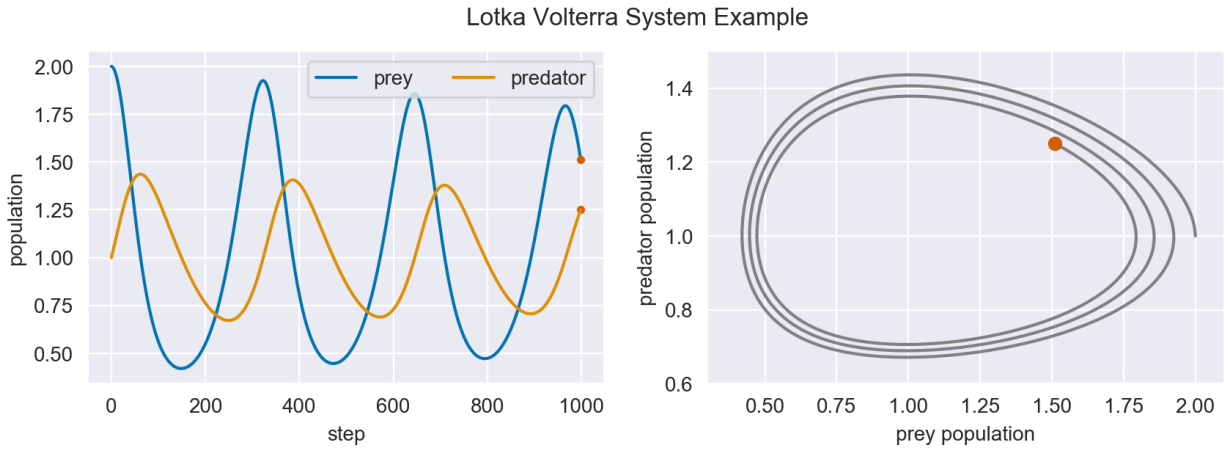
```
def f(x1, x2, alpha=4., beta=4., delta=1., gamma=1.):
    f1 = alpha*x1 - beta*x1*x2
    f2 = delta*x1*x2 - gamma*x2
    return np.array([f1, f2])
```

6.3.4 Simulation

Running this with the suite of integrators in `spacejam` then gives the following:

(s = 0) Method

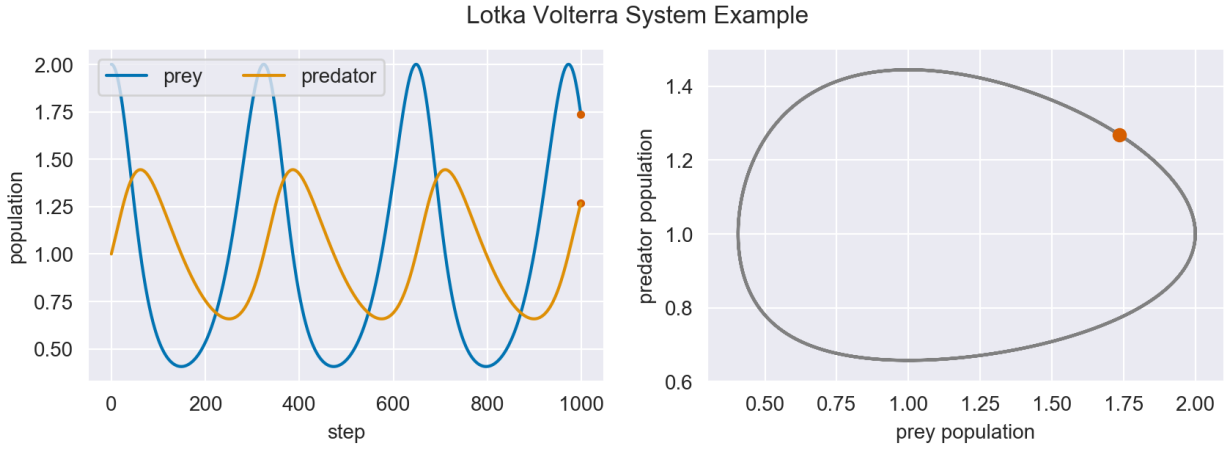
```
for n in range(N-1):
    X[n+1] = sj.integrators.amso(f, X[n], h=h, X_tol=1E-14)
```



In the plots above, we see the hallmark numerical damping of implicit schemes, which causes the overall prey and predator population to artificially decrease each step. This is especially apparent in the phase plot of the two populations where an in-spiral is present. Let's see if this is still the case for high order schemes.

(s = 1) Method

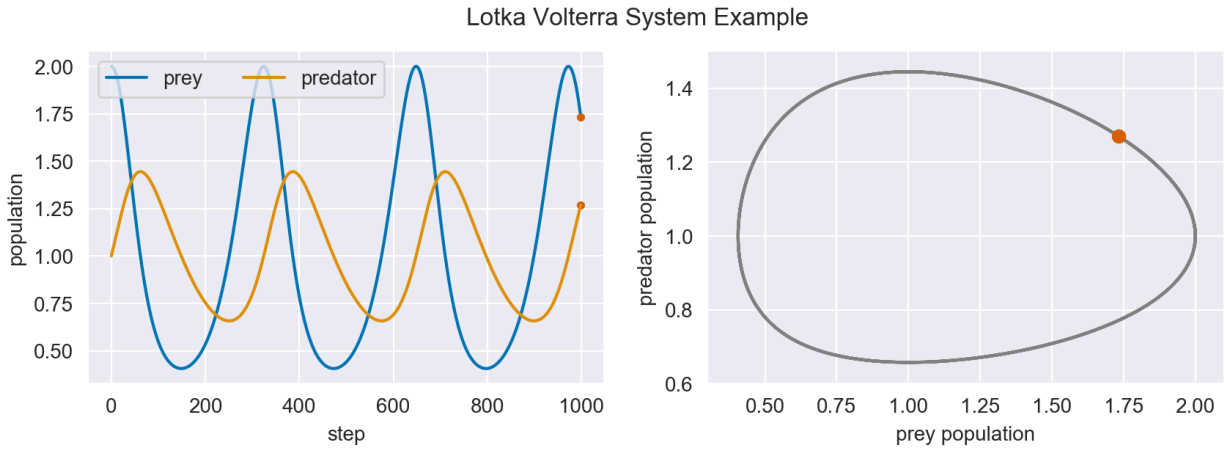
```
for n in range(N-1):
    X[n+1] = sj.integrators.amsi(f, X[n], h=h, X_tol=1E-14)
```



The spiral is gone and the ecological system is stable!

(s = 2) Method

```
for n in range(N-1):
    X[n+1] = sj.integrators.amsi(f, X[n], h=h, X_tol=1E-14)
```



As expected, the higher order scheme maintains stability as well, assuming same initial conditions. Below is an animation of the s=0 implicit simulation of this system tracking the in-spiraling of the phase plot.

Note: All plots for this example were made with the following snippet below:

```
# plot setup
sns.set_palette('colorblind')
sns.set_color_codes('colorblind')
fig, axes = plt.subplots(1, 2, figsize=(10, 3))
ax1, ax2 = axes

# solution plot
n = np.arange(N)
prey = X[:, 0]
pred = X[:, 1]

ax1.plot(n, prey, label='prey')
ax1.plot(n[-1], prey[-1], 'r.')
ax1.plot(n[-1], pred[-1], 'r.')
ax1.plot(n, pred, label='predator')
ax1.set_xlabel('step')
ax1.set_ylabel('population')
ax1.legend(ncol=2)

# phase plot
ax2.plot(prey, pred)
ax2.plot(prey[-1], pred[-1], 'r.')
ax2.set_xlabel('prey population')
ax2.set_ylabel('predator population')
ax2.set_xlim(0.3, 2.1)
ax2.set_ylim(0.6, 1.5)

plt.suptitle('Lotka Volterra System Example')
```

Movies were made with `matplotlib.animation` using its ffmeg integration. We have included a sample note-

book demoing this and the above examples in our main repo.

CHAPTER 7

Future

- Improve UI
  - Read in user-defined equations (e.g. json, yaml)
- Generalize algorithms
  - Add support for systems with time-dependent system of equations
  - Add IMEX schemes to integration suite
- Add vector support
  - Make `spacejam` object indexable so you can do stuff like this:

```
Z = sj.Dual([1, 2], [3, 4])

print(z[0], z[1])
```

```
1.00 + eps 3.00, 2.00 + eps 4.00
```

Python Module Index

S

Index

Symbols

A

C

D

E

R

S

T