desolver

Release 4.4.1

Sep 10, 2021

desolver Documentation

CHAPTER 1

DESolver

This is a python package for solving Initial Value Problems using various numerical integrators. Many integration routines are included ranging from fixed step to symplectic to adaptive integrators.

CHAPTER 2

Documentation

Documentation is now available at desolver docs! This will be updated with new examples as they are written, currently the examples show the use of `pyaudi`.

CHAPTER 3

Latest Release

4.2.0 - Improved performance of implicit methods, added embedded implicit methods following Kroulíková (2017) for fully implicit adaptive integration.

4.1.0 - Initial release of implicit integration schemes that use a basic newton-raphson algorithm to solve for the intermediate states.

3.0.0 - PyAudi support has been finalised. It is now possible to do numerical integrations using `gdual` variables such as `gdual_double`, `gdual_vdouble` and `gdual_real128` (only on select platforms, refer to pyaudi docs for more information). Install desolver with pyaudi support using `pip install desolver[pyaudi]`. Documentation has also been added and is available at desolver docs.

2.5.0 - Event detection has been added to the module. It is now possible to do numerical integration with terminal and non-terminal events.

2.2.0 - PyTorch backend is now implemented. It is now possible to numerically integrate a system of equations that use pytorch tensors and then compute gradients from these.

Use of PyTorch backend requires installation of PyTorch from here.

CHAPTER 4

To Install:

Just type

```
pip install desolver
```

4.1 Implemented Integration Methods

4.1.1 Explicit Methods

Adaptive Methods

1. Runge-Kutta 14(12) (Feagin, 2009)
2. Runge-Kutta 10(8) (Feagin, 2009)
3. Runge-Kutta 8(7) (Dormand & Prince, 1980)
4. Runge-Kutta 4(5) with Cash-Karp Coefficients
5. Adaptive Heun-Euler Method

Fixed Step Methods

1. Symplectic BABs9o7H Method (Mads & Nielsen, 2015, BAB's9o7H)
2. Symplectic ABAs5o6HA Method (Mads & Nielsen, 2015, ABAs5o6H)
3. Runge-Kutta 5 - The 5th order integrator from RK45 with Cash-Karp Coefficients.
4. Runge-Kutta 4 - The classic RK4 integrator
5. Midpoint Method
6. Heun's Method
7. Euler's Method

8. Euler-Trapezoidal Method

4.1.2 Implicit Methods [NEW]

Adaptive Methods

1. Lobatto IIIC 4(2) (Kroulíková, 2017)
2. Radau IIA 5(2) (Kroulíková, 2017)

Fixed Step Methods

1. Backward Euler
2. Implicit Midpoint
3. Crank-Nicolson
4. Lobatto IIIA 2
5. Lobatto IIIB 2
6. Lobatto IIIC 2
7. Radau IA 3
8. Radau IIA 3
9. Lobatto IIIA 4
10. Lobatto IIIB 4
11. Gauss-Legendre 4
12. Radau IA 5
13. Radau IIA 6

CHAPTER 5

Minimal Working Example

This example shows the integration of a harmonic oscillator using DESolver.

```
import desolver as de
import desolver.backend as D

def rhs(t, state, k, m, **kwargs):
    return D.array([[0.0, 1.0], [-k/m,  0.0]])@state

y_init = D.array([1., 0.])

a = de.OdeSystem(rhs, y0=y_init, dense_output=True, t=(0, 2*D.pi), dt=0.01, rtol=1e-9,
↪ atol=1e-9, constants=dict(k=1.0, m=1.0))

print(a)

a.integrate()

print(a)

print("If the integration was successful and correct, a[0].y and a[-1].y should be␣
↪near identical.")
print("a[0].y  = {}".format(a[0].y))
print("a[-1].y = {}".format(a[-1].y))

print("Maximum difference from initial state after one oscillation cycle: {}".
↪format(D.max(D.abs(a[0].y-a[-1].y))))
```

CHAPTER 6

References

Feagin, T. (2009). High-Order Explicit Runge-Kutta Methods. Retrieved from https://sce.uhcl.edu/rungekutta/

Dormand, J. R. and Prince, P. J. (1980) A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1), 19-26. https://doi.org/10.1016/0771-050X(80)90013-3

Mads, K. and Nielsen, E. (2015). *Efficient fourth order symplectic integrators for near-harmonic separable Hamiltonian systems*. Retrieved from https://arxiv.org/abs/1501.04345

Kroulíková, T. (2017). RUNGE-KUTTA METHODS (Master's thesis, BRNO UNIVERSITY OF TECHNOLOGY, Brno, Czechia). Retrieved from https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=174714

6.1 Installing desolver

desolver is a pure python module that builds upon the functionalities provided by numpy for a relatively efficient numerical integration library.

Furthermore, desolver incoporates the use of pyaudi and pytorch for more advanced applications.

With these libraries, it is possible to incorporate gradient descent of parameters into the numerical integration of some system. *Differential Intelligence* example shows how to use pyaudi to tune the weights of a controller based on the outcome of a numerical integration.

To install *desolver* simply type:

```
pip install desolver
```

Refer to the following links for pyaudi support and pytorch support. desolver will automatically detect these modules if they are in the same environment and use them if possible.

It is necessary to tell desolver if you'd like to use numpy (and pyaudi if available) or pytorch as the backend for the computations. To do this, you can set the environment variable *DES_BACKEND* to either *numpy* or *torch* as shown in the snippet below:

```
import os
os.environ['DES_BACKEND'] = 'torch' # or 'numpy'

import desolver
```

To check that all went well fire-up your python console and try the example in *quick-start example*.

6.2 Quick Start

6.2.1 My first program

If you have successfully installed desolver following the *installation guide here* you will be able to test it with the following script.

```
import desolver as de
import desolver.backend as D

@de.rhs_prettifier(
    equ_repr="[vx, -k*x/m]",
    md_repr=r"""
$$
\frac{dx}{dt} = \begin{bmatrix}
   0            & 1 \\
   -\frac{k}{m} & 0
   \end{bmatrix} \cdot \begin{bmatrix}x \\ v_x\end{bmatrix}
$$
"""
)
def rhs(t, state, k, m, **kwargs):
    return D.array([[0.0, 1.0], [-k/m,  0.0]])@state

y_init = D.array([1., 0.])

a = de.OdeSystem(rhs, y0=y_init, dense_output=True, t=(0, 2*D.pi), dt=0.01, rtol=1e-9,
↪ atol=1e-9, constants=dict(k=1.0, m=1.0))

print(a)

a.integrate()

print(a)

print("If the integration was successful and correct, a[0].y and a[-1].y should be␣
↪near identical.")
print("a[0].y  = {}".format(a[0].y))
print("a[-1].y = {}".format(a[-1].y))

print("Maximum difference from initial state after one oscillation cycle: {}".
↪format(D.max(D.abs(a[0].y-a[-1].y))))
```

Place it into a getting_started.py text file and run it with

```
python getting_started.py
```

This script shows the numerical integration of a Hooke's Law spring (harmonic oscillator) for a single cycle.

We recommend the use of Jupyter or ipython to enjoy desolver the most.

6.3 desolver Python Documentation

6.3.1 desolver package

Subpackages

desolver.exception_types package

Submodules

desolver.exception_types.exception_types module

exception desolver.exception_types.exception_types.**RecursionError**(**args*, ***kwargs*)

Bases: Exception

exception desolver.exception_types.exception_types.**FailedIntegration**(**args*, ***kwargs*)

Bases: Exception

exception desolver.exception_types.exception_types.**FailedToMeetTolerances**(**args*, ***kwargs*)

Bases: Exception

Module contents

desolver.integrators package

Submodules

desolver.integrators.integration_schemes module

desolver.integrators.integrator_template module

class desolver.integrators.integrator_template.**IntegratorTemplate**

Bases: abc.ABC

adaptive

dense_output()

get_error_estimate()

order = None

symplectic = False

update_timestep()

class desolver.integrators.integrator_template.**RichardsonIntegratorTemplate**

Bases: *desolver.integrators.integrator_template.IntegratorTemplate*, abc.ABC

adaptive

symplectic = False

desolver.integrators.integrator_types module

class desolver.integrators.integrator_types.**RungeKuttaIntegrator**(*sys_dim*, *dtype=None*, *rtol=None*, *atol=None*, *device=None*)

Bases: desolver.integrators.integrator_types.TableauIntegrator

adaptive

algebraic_system(*next_state*, *rhs*, *initial_time*, *initial_state*, *timestep*, *constants*)

algebraic_system_jacobian(*next_state*, *rhs*, *initial_time*, *initial_state*, *timestep*, *constants*)

dense_output()

explicit

explicit_stages

final_state = None

fsal

get_error_estimate()

implicit

implicit_stages

classmethod is_adaptive()

classmethod is_explicit()

classmethod is_fsal()

classmethod is_implicit()

step(*rhs*, *initial_time*, *initial_state*, *constants*, *timestep*)

update_timestep()

class desolver.integrators.integrator_types.**ExplicitSymplecticIntegrator**(*sys_dim*, *dtype=None*, *staggered_mask=None*, *rtol=None*, *atol=None*, *device=None*)

Bases: desolver.integrators.integrator_types.TableauIntegrator

A base class for all symplectic numerical integration methods.

A ExplicitSymplecticIntegrator derived object corresponds to a numerical integrator tailored to a particular dynamical system with an integration scheme defined by the sequence of drift-kick coefficients in tableau.

An explicit symplectic integrator may be considered as a sequence of carefully picked drift and kick stages that build off the previous stage which is the implementation considered here. A masking array of indices indicates the drift and kick variables that are updated at each stage.

In a system defined by a Hamiltonian of q and p (generalised position and generalised momentum respectively), the drift stages update q and the kick stages update p. For a conservative Hamiltonian, a symplectic method will minimise the drift in the Hamiltonian during the integration.

tableau
A numpy array with N stages and 3 entries per stage where the first column is the timestep fraction and the remaining columns are the drift/kick coefficients.

Type numpy array, shape (N, N+1)

symplectic
True if the method is symplectic.

Type bool

adaptive

dense_output()

explicit

explicit_stages

fsal

implicit

implicit_stages

classmethod is_adaptive()

classmethod is_explicit()

classmethod is_fsal()

classmethod is_implicit()

step(*rhs*, *initial_time*, *initial_state*, *constants*, *timestep*)

symplectic = True

desolver.integrators.integrator_types.**generate_richardson_integrator**(*basis_integrator*)
A function for generating an integrator that uses local Richardson Extrapolation to find the change in state ΔY over a timestep h by estimating lim ΔY as h->0.

Takes any integrator as input and returns a specialisation of the RichardsonExtrapolatedIntegrator class that uses basis_integrator as the underlying integration mechanism.

Parameters **basis_integrator** (*A subclass of IntegratorTemplate or a class that implements the methods and attributes of IntegratorTemplate.*) –

Returns returns the Richardson Extrapolated specialisation of basis_integrator

Return type RichardsonExtrapolatedIntegrator

Module contents

desolver.integrators.**available_methods**(*names=True*)

desolver.utilities package

Submodules

desolver.utilities.interpolation module

class desolver.utilities.interpolation.**CubicHermiteInterp**(*t0*, *t1*, *p0*, *p1*, *m0*, *m1*)

Bases: object

Cubic Hermite Polynomial Interpolation Class

Constructs a cubic Hermite polynomial interpolant for a function with values p0 and p1, and gradients m0 and m1 at t0 and t1 respectively.

Parameters

- **t1** (*t0,*) – Evaluation points
- **p1** (*p0,*) – Function values at t0 and t1
- **m1** (*m0,*) – Function gradients wrt. t and t0 and t1

trange

tshift

desolver.utilities.optimizer module

desolver.utilities.optimizer.**brentsroot**(*f*, *bounds*, *tol=None*, *verbose=False*, *return_interval=False*)

Brent's algorithm for finding root of a bracketed function.

Parameters

- **f** (*callable*) – callable that evaluates the function whose roots are to be found
- **bounds** (*tuple of float, shape (2,)*) – lower and upper bound of interval to find root in
- **tol** (*float-type*) – numerical tolerance for the precision of the root
- **verbose** (*bool*) – set to true to print useful information

Returns returns the location of the root if found and a bool indicating a root was found

Return type tuple(float-type, bool)

Examples

```
>>> def ft(x):
    return x**2 - (1 - x)**5
>>> xl, xu = 0.1, 1.0
>>> x0, success = brentsroot(ft, xl, xu, verbose=True)
>>> success, x0, ft(x0)
(True, 0.34595481584824206, 6.938893903907228e-17)
```

desolver.utilities.optimizer.**brentsrootvec**(*f*, *bounds*, *tol=None*, *verbose=False*, *return_interval=False*)

Vectorised Brent's algorithm for finding root of bracketed functions.

Parameters

- **f** (*list of callables*) – list of callables each of which evaluates the function to find the root of
- **bounds** (*tuple of float, shape (2,)*) – lower and upper bound of interval to find root in
- **tol** (*float-type*) – numerical tolerance for the precision of the roots
- **verbose** (*bool*) – set to true to print useful information

Returns returns a list of the locations of roots and a list of bools indicating whether or not a root was found in the interval

Return type tuple(list(float-type), list(bool))

Examples

```
>>> f = lambda x: lambda y: x * y - y**2 + x
>>> xl, xu = 0.1, 1.0
>>> funcs = [f(i*0.5) for i in range(3)]
>>> x0, success = brentsrootvec(funcs, xl, xu, verbose=True)
>>> success, x0, [funcs[i](x0[i]) for i in range(len(funcs))]
(array([ True,  True,  True]), array([0.        , 1.        , 1.61803399]), [0.0,␣
↪0.0, 0.0])
```

desolver.utilities.optimizer.**newtontrustregion**(*f, x0, jac=None, tol=None, verbose=False, maxiter=200, jac_update_rate=20, initial_trust_region=None*)

desolver.utilities.optimizer.**nonlinear_roots**(*f, x0, jac=None, tol=None, verbose=False, maxiter=200, use_scipy=True, additional_args=(), additional_kwargs={}*)

desolver.utilities.utilities module

class desolver.utilities.utilities.**JacobianWrapper**(*rhs, base_order=2, richardson_iter=None, adaptive=True, flat=False, atol=None, rtol=None*)

Bases: object

A wrapper class that uses Richardson Extrapolation and 4th order finite differences to compute the jacobian of a given callable function.

The Jacobian is computed using up to 16 richardson iterations which translates to a maximum order of 4+16 for the gradients with respect to Δx as Δx->0. The evalaution is adaptive so within the given tolerances, the evaluation will exit early in order to minimise computation so gradients can be calculated with controllable precision.

rhs

A callable function that

Type Callable of the form f(x)->D.array of arbitrary shape

base_order
The order of the underlying finite difference gradient estimate, this can be evaluated at different step sizes using the *estimate* method.

Type int

richardson_iter
The maximum number of richardson iterations to use for estimating gradients. In most cases, less than 16 should be enough, and if you start needing more than 16, it might be worth considering that finite differences are inappropriate.

Type int

order
Maximum order of the gradient estimate

Type int

adaptive
Whether to use adaptive evaluation of the richardson extrapolation or to run for the full 16 iterations.

Type bool

atol, rtol
Absolute and relative tolerances for the adaptive gradient extrapolation

Type float

flat
If set to True, the gradients will be returned as a ravelled array (ie. jacobian.shape = (-1,)) If set to False, the shape will be (*x.shape, *f(x).shape). When x and f(x) are vector valued of length n and m respectively, the gradient will have the shape (n,m) as expected of the Jacobian of f(x).

Type bool

adaptive_richardson(*y*, **args*, *dy=0.5*, *factor=4*, ***kwargs*)

check_converged(*initial_state*, *diff*, *prev_error*)

estimate(*y*, **args*, *dy=None*, ***kwargs*)

static finite_difference_weights(*number_of_nodes*, *order=1*)

richardson(*y*, **args*, *dy=0.5*, *factor=4.0*, ***kwargs*)

desolver.utilities.utilities.**convert_suffix**(*value*, *suffixes=('d', 'h', 'm', 's')*, *ratios=(24, 60, 60)*, *delimiter=':'*)

Converts a base value into a human readable format with the given suffixes and ratios.

Parameters

- **value** (*int or float*) – value to be converted
- **suffixes** (*list of str*) – suffixes for each subdivision (eg. ['days', 'hours', 'minutes', 'seconds'])
- **ratios** (*list of int*) – the relative period of each subdivision (eg. 24 hours in 1 day, 60 minutes in 1 hour, 60 seconds in 1 minute -> [24, 60, 60])
- **delimiter** (*str*) – string to use between each subdivision

Returns returns string with the subdivision values and suffixes joined together

Return type str

Examples

```
>>> convert_suffix(3661, suffixes=['d', 'h', 'm', 's'], ratios=[24, 60, 60],␣
↪delimiter=':')
'0d:1h:1m1.00s'
```

desolver.utilities.utilities.**warning**(*message*, *category=<class 'Warning'>*)

Convenience function for printing to sys.stderr.

Parameters

- **message** (*str*) – warning message
- **category** (*warning category*) – type of warning message. eg. DeprecationWarning

Examples

```
>>> warning("Things have failed...", warning.Warning)
Things have failed...
```

desolver.utilities.utilities.**search_bisection**(*array*, *val*)

Finds the index of the nearest value to val in array. Uses the bisection method.

Parameters

- **array** (*list of numeric values*) – list to search, assumes the list is sorted (will not work if it isn't sorted!)
- **val** (*numeric*) – numeric value to find the nearest value in the array.

Returns returns the index of the position in the array with the value closest to val

Return type int

Examples

```
>>> list_to_search = [1,2,3,4,5]
>>> val_to_find    = 2.5
>>> idx = search_bisection(list_to_search, val_to_find)
>>> idx, list_to_search[idx]
(1, 2)
```

desolver.utilities.utilities.**search_bisection_vec**(*array*, *val*)

Finds the indices of the nearest values in array to each val. Uses the bisection method.

Parameters

- **array** (*array of numeric values*) – list to search, assumes the list is sorted (will not work if it isn't sorted!)
- **val** (*array of numeric values*) – numeric values to find the nearest value in array.

Returns returns the indices of the positions in the array with the value closest to val

Return type array(int)

Examples

```
>>> list_to_search = [1,2,3,4,5]
>>> val_to_find    = [1.5,3.5]
>>> idx = search_bisection(list_to_search, val_to_find)
>>> idx, list_to_search[idx]
([0,2], [1,3])
```

class desolver.utilities.utilities.**BlockTimer**(*section_label=None*, *start_now=True*, *suppress_print=False*)

Bases: object

Timing Class

Takes advantage of the with syntax in order to time a block of code.

Parameters

- **section_label** (*str*) – name given to section of code
- **start_now** (*bool*) – if True the timer is started upon construction, otherwise start() must be called.
- **suppress_print** (*bool*) – if True a message will be printed upon destruction with the section_label and the code time.

elapsed()

Method to get the elapsed time.

Returns Returns the elapsed time since timer start if stop() was not called, otherwise returns the elapsed time between timer start and when elapsed() is called.

Return type float

end()

Method to stop the timer

restart_timer()

Method to restart the timer.

Sets the start time to now and resets the end time.

start()

Method to start the timer

Module contents

Submodules

desolver.differential_system module

class desolver.differential_system.**DiffRHS**(*rhs*, *equ_repr=None*, *md_repr=None*)

Bases: object

Differential Equation class. Designed to wrap around around a function for the right-hand side of an ordinary differential equation.

rhs

Right-hand side of an ordinary differential equation

Type callable

equ_repr
String representation of the right-hand side.

Type str

hook_jacobian_call(*jac_fn*)
Attaches a function, jac_fn, that returns the jacobian of self.rhs as an array with shape (state.numel(), rhs.numel()).

jac(*t*, *y*, **args*, ***kwargs*)
Returns the jacobian of self.rhs at a given time (t) and state (y). Tracks number of jacobian evaluations.

Uses a Richardson Extrapolated 5th order central finite differencing method when a jacobian is not defined as self.rhs.jac or by self.hook_jacobian_call

jac_is_wrapped_rhs

set_jac_base_order(*order*)

unhook_jacobian_call()
Detaches the jacobian function and replaces it with a finite difference estimate if a jacobian function was originally attached.

desolver.differential_system.**rhs_prettifier**(*equ_repr=None*, *md_repr=None*)

class desolver.differential_system.**OdeSystem**(*equ_rhs*, *y0*, *t=(0, 1)*, *dense_output=False*, *dt=1.0*, *rtol=None*, *atol=None*, *constants={}*)

Bases: object

Ordinary Differential Equation class. Designed to be used with a system of ordinary differential equations.

atol
The absolute tolerance of the adaptive integration schemes

constants
A dictionary of constants for the differential system.

dt
The timestep of the numerical integration

events
A tuple of (time, state) tuples at which each event occurs.

Examples

```
>>> ode_system = desolver.OdeSystem(...)
>>> ode_system.integrate(events=[...])
>>> ode_system.events
(StateTuple(t=..., y=...), StateTuple(t=..., y=...), StateTuple(t=..., y=...),
↪ ...)
```

get_current_time()
Returns the current time of the ODE system

get_step_interpolant()

initialise_integrator(*preserve_states=False*)

integrate(*t=None*, *callback=None*, *eta=False*, *events=None*)
Integrates the system to a specified time.

Parameters

- **t** (*float*) – If t is specified, then the system will be integrated to time t. Otherwise the system will integrate to the specified final time. NOTE: t can be negative in order to integrate backwards in time, but use this with caution as this functionality is slightly unstable.
- **callback** (*callable or list of callables*) – A callable object or list of callable objects that are invoked as callback(self) at each time step. e.g. for logging integration to disk, saving data, manipulating the state of the system, etc.
- **eta** (*bool*) – Specifies whether or not the integration process should return an eta, current progress and simple information regarding step-size and current time. Will be deprecated in the future in favour of verbosity argument that prints once every n-steps. NOTE: This may slow the integration process down as the process of outputting these values create overhead.
- **events** (*callable or list of callables*) – Events to track, defaults to None. Each function must have the signature `event(t, y, **kwargs)` and the solver will find the time t such that `event(t, y, **kwargs) == 0`. The `**kwargs` argument allows the solver to pass the system constants to the function. Additionally, each event function can possess the following two attributes:

  direction: bool, optional Indicates the direction of the event crossing that will register an event.

  is_terminal: bool, optional Indicates whether the detection of the event terminates the numerical integration.

Raises *RecursionError :* – Raised if an adaptive integrator recurses beyond the recursion limit when attempting to compute a forward step. This usually means that the numerical integration did not converge and that the behaviour of the system is highly unreliable. This could be due to numerical issues.

integration_status

Returns the integration status as a human-readable string.

Returns String containing the integration status message.

Return type str

method

The numerical integration scheme

nfev

The number of function evaluations used during the numerical integration

njev

The number of jacobian evaluations used during the numerical integration

reset()

Resets the system to the initial time.

rtol

The relative tolerance of the adaptive integration schemes

set_kick_vars(*staggered_mask*)

Sets the variable mask for the symplectic integrators.

The conventional structure of a symplectic integrator is Kick-Drift-Kick or Drift-Kick-Drift where Drift is when the Positions are updated and Kick is when the Velocities are updated.

The default assumption is that the latter half of the variables are to be updated in the Kick step and the former half in the Drift step.

Does nothing if integrator is not symplectic.

Parameters **staggered_mask** (*array of bools*) – A boolean array with the same shape as y. Specifies the elements of y that are to be updated as part of the Kick step.

set_method(*new_method*, *staggered_mask=None*, *preserve_states=True*)

Sets the method of integration.

Parameters

- **new_method** (*str or stateful functor*) – A string that is the name of an integrator that is available in DESolver, OR a functor that takes the current state, time, time step and equation, and advances the state by 1 timestep adjusting the timestep as necessary.
- **staggered_mask** (*boolean masking array*) – A boolean array with the same shape as the system state that indicates which variables are updated during the 'kick' stage of a symplectic integrator. Has no effect if the integrator is adaptive.

Raises `ValueError` – If the string is not a valid integration scheme.

sol

success

t

The times at which the system has been evaluated.

t0

The initial integration time

tf

The final integration time

y

The states at which the system has been evaluated.

Module contents

6.4 Using numpy

6.4.1 The Simple Harmonic Oscillator

Here we will expand on the harmonic oscillator first shown in the getting started script. I'll walk you through some of the features of desolver and hopefully give a better a sense of how to use the software.

So let's begin!

First we import the libraries we'll need. I import all the matplotlib machinery using the magic command `%matplotlib`, but this is only for notebook/ipython environments.

Then I import `desolver` and the desolver backend as well (this will be useful for specifying our problem), and set the default datatype to `float64`.

[1]:
```
%matplotlib inline
from matplotlib import pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import desolver as de
import desolver.backend as D

D.set_float_fmt('float64')
```

```
Using numpy backend
```

Specifying the Dynamical System

Now let's specify the right hand side of our dynamical system. It should be

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = -\frac{k}{m}x$$

But desolver only works with first order differential equations, thus we must cast this into a first order system before we can solve it. Thus we obtain the following system

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= v_x \\ \frac{\mathrm{d}v_x}{\mathrm{d}t} &= -\frac{k}{m}x\end{aligned}$$

which can be specified as a simple matrix equation as

$$\begin{aligned}\frac{\mathrm{d}y}{\mathrm{d}t} &= \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & 0 \end{bmatrix} \cdot \vec{y} \\ \vec{y} &= \begin{bmatrix} x \\ v_x \end{bmatrix}\end{aligned}$$

[2]:
```
@de.rhs_prettifier(
    equ_repr="[vx, -k*x/m]",
    md_repr=r"""
$$
\frac{\mathrm{d}y}{\mathrm{dt}} = \begin{bmatrix}
   0            & 1 \\
   -\frac{k}{m} & 0
   \end{bmatrix} \cdot \vec y
$$
"""
)
def rhs(t, state, k, m, **kwargs):
    return D.array([[0.0, 1.0], [-k/m,  0.0]])@state
```

First thing to notice is that we used the backend to specify the matrix and minimise the use of numpy specific machinery. This isn't necessary if you only use numpy, but by doing this we can make this code run with the pytorch backend with minimal effort.

Second thing is the use of the decorator `@de.rhs_prettifier`, this is a convenience decorator that allows me to specify a text representation of the differential equations. Convenient if I want to print it

[3]:
```
print(rhs)
```

```
[vx, -k*x/m]
```

Or if I want it to look pretty when it is rendered in the notebook

[4]:
```
display(rhs)
```

$$\frac{\mathrm{d}y}{\mathrm{dt}} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & 0 \end{bmatrix} \cdot \vec{y}$$

Let's specify the initial conditions as well

[5]:
```
y_init = D.array([1., 0.])
```

And now we're ready to integrate!

The Numerical Integration

There are a number of things we must choose before we numerically integrate our system of equations. The first of these is whether or not we want an interpolating spline so that we can compute the state of our system between timesteps. The second is the duration of the numerical integration. And the third is the value of parameters of the system: `k` and `m`.

Unlike scipy, desolver let's you specify a dictionary of constants that are passed to the `rhs` function and can be modified even after constructing the `OdeSystem` object. This is particularly useful if you want to vary a single constant over multiple integrations without changing any other parameters.

Now, we'll set the numerical integration to 1 cycle of the oscillator at `k=1` and `m=1` which, when computed from the formula $T = 2\pi\sqrt{\frac{k}{m}}$, is exactly 2π.

[6]:
```
a = de.OdeSystem(rhs, y0=y_init, dense_output=True, t=(0, 2*D.pi), dt=0.01, rtol=1e-9,
↪ atol=1e-9, constants=dict(k=1.0, m=1.0))
```

[7]:
```
a.integrate()
```

Since `k=1` and `m=1` and we integrated for 1 cycle, we expect that the final state of the system is the same as the initial state.

[8]:
```
print("initial state = {}".format(a[0].y))
print("final state   = {}".format(a[-1].y))
print("maximum absolute difference = {}".format(D.max(D.abs(a[-1].y - a[0].y))))
```

```
initial state = [1. 0.]
final state   = [ 1.00000000e+00 -1.14580831e-10]
maximum absolute difference = 3.2781628522826622e-09
```

Wonderful! We see that the final state is almost exactly the same. Furthermore, we see that this is within the tolerances we specified when creating the `OdeSystem` where we set `rtol` and `atol` to `1e-9`.

To show you that this is not a fluke, we'll change them to `1e-12` and see what happens.

[9]:
```
a.rtol = 1e-12
a.atol = 1e-12
a.reset()
a.integrate()
```

[10]:
```
print("initial state = {}".format(a[0].y))
print("final state   = {}".format(a[-1].y))
print("maximum absolute difference = {}".format(D.max(D.abs(a[-1].y - a[0].y))))
```

```
initial state = [1. 0.]
final state   = [ 1.00000000e+00 -2.88935542e-14]
maximum absolute difference = 3.255395952805884e-12
```

It's very simple to change the tolerances and rerun the system. Furthermore, we can update our constants and see what happens.

If we quadruple `k`, the spring constant, the period will double, and so after an integration period of 2π the system should, yet again, be in a final state that is almost exactly the initial state.

[11]:
```
a.constants['k'] = a.constants['k'] * 4
a.reset()
a.integrate()
```

[12]:
```
print("initial state = {}".format(a[0].y))
print("final state   = {}".format(a[-1].y))
print("maximum absolute difference = {}".format(D.max(D.abs(a[-1].y - a[0].y))))
```

```
initial state = [1. 0.]
final state   = [ 1.00000000e+00 -1.69420034e-13]
maximum absolute difference = 9.035661108214299e-12
```

The final state is again almost the same as the initial state, but now the maximum absolute difference has increased. This is due to the fact that the numerical error when using an adaptive runge-kutta method is not a random walk, but a function of the whole numerical procedure. Thus if we double the initial integration time, and set `k=1` again, we'll see that the error is larger.

[13]:
```
a.constants['k'] = 1
a.tf = 4*D.pi
a.reset()
a.integrate()
```

[14]:
```
print("initial state = {}".format(a[0].y))
print("final state   = {}".format(a[-1].y))
print("maximum absolute difference = {}".format(D.max(D.abs(a[-1].y - a[0].y))))
```

```
initial state = [1. 0.]
final state   = [ 1.00000000e+00 -5.68208675e-14]
maximum absolute difference = 6.517453243759519e-12
```

The longer we integrate for, the larger this error will become. Is there anything we can do?

YES

We can use a symplectic integrator since this is a system with a Hamiltonian $H = \frac{kx^2}{2} + \frac{mv_x^2}{/}2$.

A symplectic integrator preserves the symplectic two-form $\mathrm{d}\vec{p} \wedge \mathrm{d}\vec{q}$ where p is the momentum and q is the position such that

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -\frac{\partial H}{\partial q}$$
$$\frac{\mathrm{d}q}{\mathrm{d}t} = \frac{\partial H}{\partial p}$$

where, in our case, $v_x = \frac{p}{m}$ and $x = q$.

Why is this important? I'll leave the detailed theory to Wikipedia and other sources, but the jist of it is that a symplectic integrator is essentially a geometric transformation in the phase space of the system and thus preserves the differential volume element of a Hamiltonian that is almost, but not quite, the Hamiltonian of the system.

This is great because it means that, in the best case scenario, the errors in the numerically integrated states are random walks instead of increasing linearly with the integration time. The downside is that a symplectic integrator is not adaptive and thus requires more function evaluations than a Runge-Kutta method.

[15]:
```
a.set_method("BABS9O7H")
a.dt = 0.05
a.tf = 2*D.pi
a.integrate()
```

```
An integration was already run, the system will be reset
```

[16]:
```
print("initial state = {}".format(a[0].y))
print("final state   = {}".format(a[-1].y))
print("maximum absolute difference = {}".format(D.max(D.abs(a[-1].y - a[0].y))))
```

```
initial state = [1. 0.]
final state   = [ 1.0000000e+00 -2.7269853e-15]
maximum absolute difference = 2.7269853042355408e-15
```

[17]:
```
a.set_method("BABS9O7H")
a.dt = 0.05
a.tf = 8*D.pi
a.integrate()
```

```
An integration was already run, the system will be reset
```

[18]:
```
print("initial state = {}".format(a[0].y))
print("final state   = {}".format(a[-1].y))
print("maximum absolute difference = {}".format(D.max(D.abs(a[-1].y - a[0].y))))
```

```
initial state = [1. 0.]
final state   = [ 1.00000000e+00 -9.48546797e-15]
maximum absolute difference = 9.485467966641181e-15
```

[19]:
```
a.set_method("BABS9O7H")
a.dt = 0.05
a.tf = 32*D.pi
a.integrate()
```

```
An integration was already run, the system will be reset
```

[20]:
```
print("initial state = {}".format(a[0].y))
print("final state   = {}".format(a[-1].y))
print("maximum absolute difference = {}".format(D.max(D.abs(a[-1].y - a[0].y))))
```

```
initial state = [1. 0.]
final state   = [ 1.00000000e+00 -4.41313652e-14]
maximum absolute difference = 4.413136522884997e-14
```

Above, I've run the numerical integration using a step size of 0.05 for increasing integration periods from one cycle to four cycles to sixteen cycles and, despite that, the error has stayed near the limits of double precision arithmetic. If I further integrate for 1024 cycles, we'll see that the error begins to increase and this is expected because although the errors in each step may be random walks, they have a cumulative effect that is not necessarily a random walk.

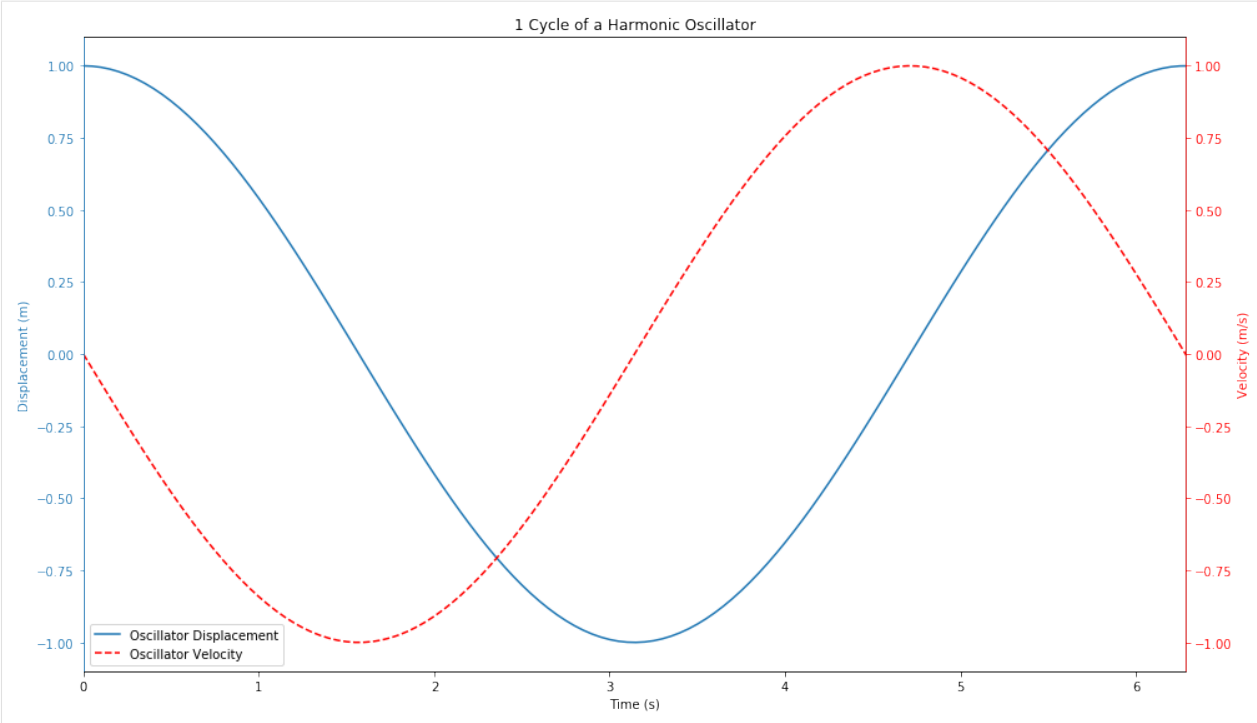
[21]:
```
a.set_method("BABS9O7H")
a.dt = 0.05
a.tf = 2*1024*D.pi
a.integrate()
```

```
An integration was already run, the system will be reset
```

[22]:
```
print("initial state = {}".format(a[0].y))
print("final state   = {}".format(a[-1].y))
print("maximum absolute difference = {}".format(D.max(D.abs(a[-1].y - a[0].y))))
```

```
initial state = [1. 0.]
final state   = [ 1.00000000e+00 -2.06966388e-12]
maximum absolute difference = 2.069663884718409e-12
```

[23]:
```
fig = plt.figure(figsize=(14,8))
ax  = fig.add_subplot(111)

displn = ax.plot(a.t, a.y[:, 0], label="Oscillator Displacement", color='C0')
axt = ax.twinx()
velln  = axt.plot(a.t, a.y[:, 1], label="Oscillator Velocity", color='red', linestyle=
↪'--')

ax.set_xlabel("Time (s)")
ax.set_ylabel("Displacement (m)")
axt.set_ylabel("Velocity (m/s)")
ax.set_xlim(0, 2*D.pi)

ax.spines['left'].set_color('C0')
ax.tick_params(axis='y', colors='C0')
ax.yaxis.label.set_color('C0')

axt.spines['right'].set_color('red')
axt.spines['left'].set_color('C0')
axt.tick_params(axis='y', colors='red')
axt.yaxis.label.set_color('red')

# added these three lines
lns = displn + velln
labs = [l.get_label() for l in lns]
ax.legend(lns, labs)

ax.set_title("1 Cycle of a Harmonic Oscillator")
plt.tight_layout()
```



Looking at the Hamiltonian

So we've said that a symplectic integrator preserves a perturbed Hamiltonian, we should be able to see this by computing the Hamiltonian at each timestep and looking at how it evolves over the course of a numerical integration.

Let's first define the Hamiltonian.

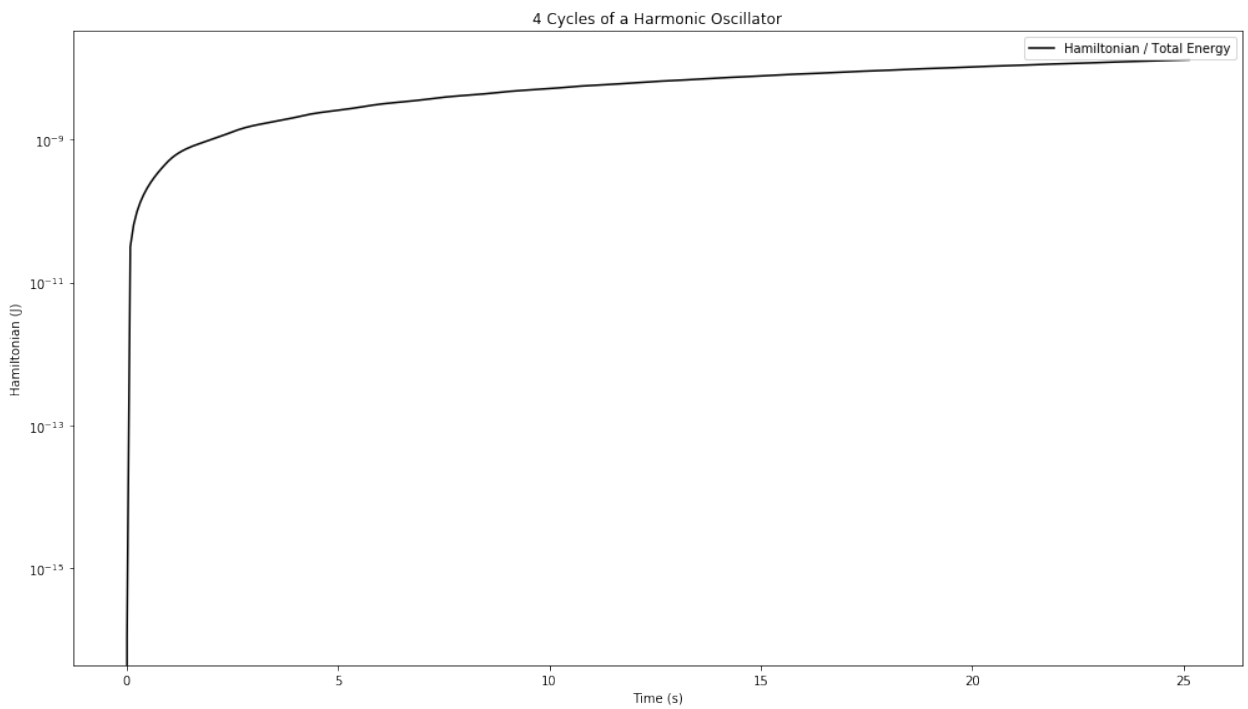
[24]:
```
def kinetic_energy(t, state, k, m):
    x, vx = state
    return m * vx**2 / 2

def potential_energy(t, state, k, m):
    x, vx = state
    return k * x**2 / 2

def hamiltonian(t, state, k, m):
    return kinetic_energy(t, state, k, m) + potential_energy(t, state, k, m)
```

Now it's not interesting to just look at the Hamiltonian alone, we'd like to look at **how** the Hamiltonian evolves so we will look at the absolute difference between the Hamiltonian at time t and the initial Hamiltonian.

To start off, we'll look at the Hamiltonian when we use an adaptive integrator.

[25]:
```
a.reset()
a.method = "RK45"
a.rtol = 1e-9
a.atol = 1e-9
a.tf   = 4*2*D.pi
a.integrate()
```

[26]:
```
fig = plt.figure(figsize=(14,8))
ax  = fig.add_subplot(111)

E_H = D.abs(hamiltonian(a.t, a.y.T, **a.constants) - hamiltonian(a.t[0], a.y[0], **a.
↪constants))

disp_H = ax.plot(a.t, E_H, label="Hamiltonian / Total Energy", color='black')

ax.set_xlabel("Time (s)")
ax.set_ylabel("Hamiltonian (J)")

# added these three lines
ax.legend()
ax.set_yscale("log")

ax.set_title("{:.0f} Cycle{} of a Harmonic Oscillator".format(a.tf/(2*D.pi), "s" if a.
↪tf/(2*D.pi) > 1 else ""))
plt.tight_layout()
```

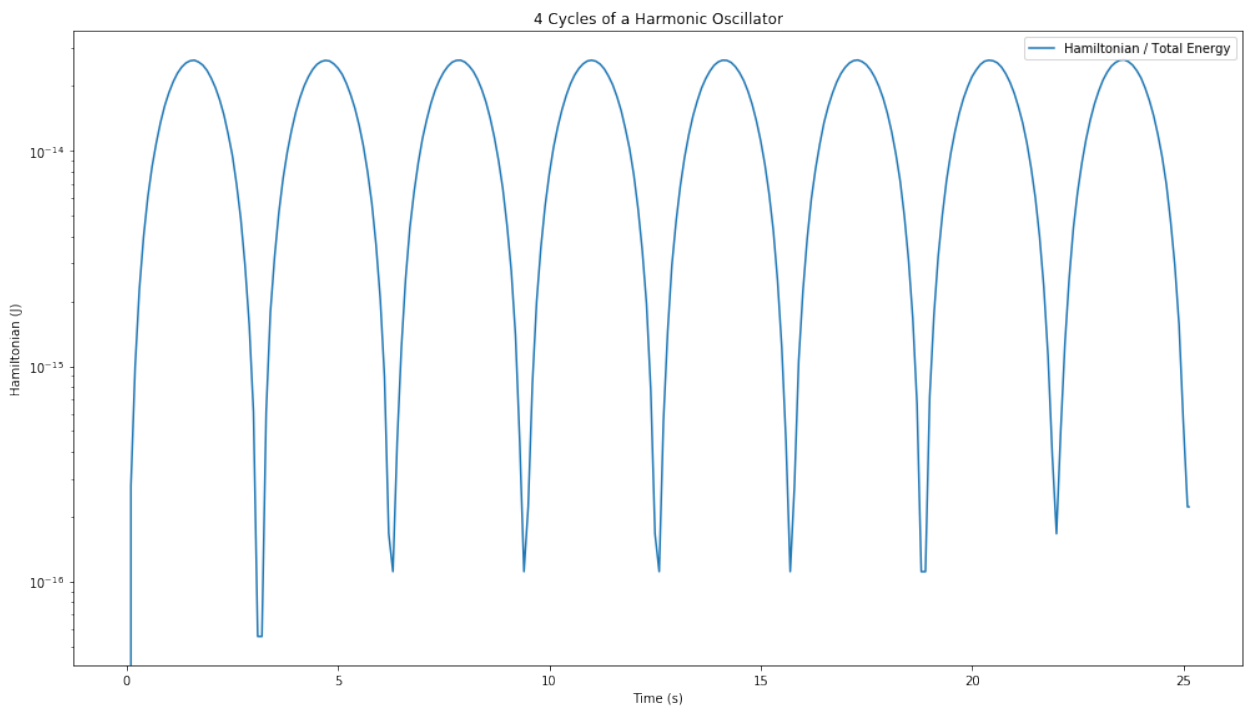


We see that the Hamiltonian starts off correctly, but then, very rapidly, jumps up to an error on the order of 10^{-9} which is the same as the tolerance we've set for the numerical integration.

Now let's compare this to a symplectic integrator.

[27]:
```
a.reset()
a.set_method("BABS9O7H")
a.rtol = 1e-9
a.atol = 1e-9
a.dt   = 1e-1
a.tf   = 4*2*D.pi
a.integrate()
```

[28]:
```
fig = plt.figure(figsize=(14,8))
ax  = fig.add_subplot(111)

E_H = D.abs(hamiltonian(a.t, a.y.T, **a.constants) - hamiltonian(a.t[0], a.y[0], **a.
↪constants))

disp_H = ax.plot(a.t, E_H, label="Hamiltonian / Total Energy", color='C0')

ax.set_xlabel("Time (s)")
ax.set_ylabel("Hamiltonian (J)")

# added these three lines
ax.legend()
ax.set_yscale("log")

ax.set_title("{:.0f} Cycle{} of a Harmonic Oscillator".format(a.tf/(2*D.pi), "s" if a.
↪tf/(2*D.pi) > 1 else ""))
plt.tight_layout()
```



Well that's completely different behaviour to the adaptive integrator. We see now that the Hamiltonian oscillates between a maximal error and a minimal one, but remains within 10^{-14} of the true Hamiltonian. This is exactly the behaviour we were expecting. Let's look further long term and compare with the adaptive integrator, we'll decrease the adaptive integration tolerances down to 10^{-12} to see if that helps.

[29]:
```
fig = plt.figure(figsize=(14,8))
ax  = fig.add_subplot(111)

a.tf = 1024*2*D.pi

a.reset()
a.set_method("BABS9O7H")
a.dt   = 1e-1
```

(continues on next page)

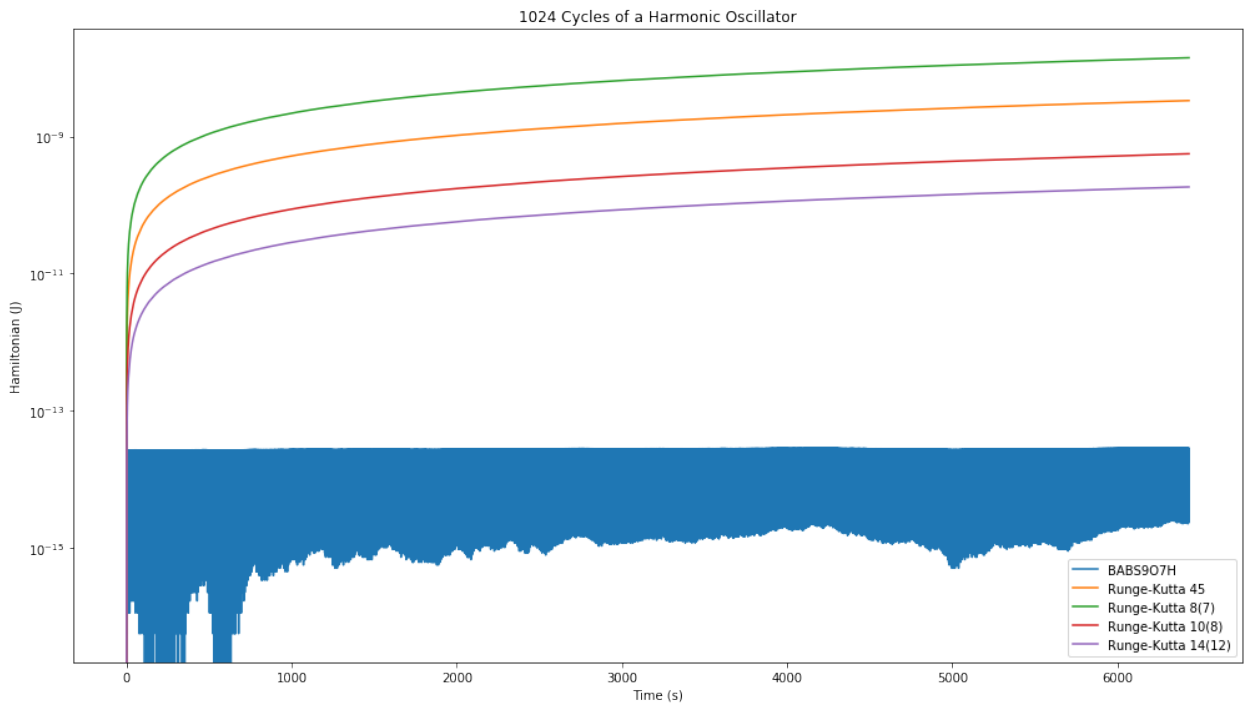
(continued from previous page)

```
a.integrate()

E_H = D.abs(hamiltonian(a.t, a.y.T, **a.constants) - hamiltonian(a.t[0], a.y[0], **a.
↪constants))

disp_H = ax.plot(a.t, E_H, label="BABS9O7H", color='C0')

a.reset()
a.set_method("RK45")
a.rtol = 1e-12
a.atol = 1e-12
a.dt   = 1e-1
a.integrate()

E_H = D.abs(hamiltonian(a.t, a.y.T, **a.constants) - hamiltonian(a.t[0], a.y[0], **a.
↪constants))

disp_H = ax.plot(a.t, E_H, label="Runge-Kutta 45", color='C1')

a.reset()
a.set_method("RK87")
a.rtol = 1e-12
a.atol = 1e-12
a.dt   = 1e-1
a.integrate()

E_H = D.abs(hamiltonian(a.t, a.y.T, **a.constants) - hamiltonian(a.t[0], a.y[0], **a.
↪constants))

disp_H = ax.plot(a.t, E_H, label="Runge-Kutta 8(7)", color='C2')

a.reset()
a.set_method("RK108")
a.rtol = 1e-12
a.atol = 1e-12
a.dt   = 1e-1
a.integrate()

E_H = D.abs(hamiltonian(a.t, a.y.T, **a.constants) - hamiltonian(a.t[0], a.y[0], **a.
↪constants))

disp_H = ax.plot(a.t, E_H, label="Runge-Kutta 10(8)", color='C3')

a.reset()
a.set_method("RK1412")
a.rtol = 1e-12
a.atol = 1e-12
a.dt   = 1e-1
a.integrate()

E_H = D.abs(hamiltonian(a.t, a.y.T, **a.constants) - hamiltonian(a.t[0], a.y[0], **a.
↪constants))

disp_H = ax.plot(a.t, E_H, label="Runge-Kutta 14(12)", color='C4')

ax.set_xlabel("Time (s)")
ax.set_ylabel("Hamiltonian (J)")
```

(continues on next page)

(continued from previous page)

```
# added these three lines
ax.legend(loc='lower right')
ax.set_yscale("log")

ax.set_title("{:.0f} Cycle{} of a Harmonic Oscillator".format(a.tf/(2*D.pi), "s" if a.
↪tf/(2*D.pi) > 1 else ""))
plt.tight_layout()
```



That's not good, the Hamiltonian error increases consistently with continued integration whereas the 7th order symplectic integrator continues to stay bounded.

Thus we see that if energy conservation is a concern, then a symplectic integrator should be used. In most cases, the adaptive Runge-Kutta methods are more useful as they generally require fewer function evaluations and give very good results.

6.4.2 The Duffing Oscillator

In this notebook we will explore the Duffing Oscillator and attempt to recreate the time traces and phase portraits shown on the Duffing Oscillator Wikipedia page

[1]:
```
%matplotlib inline
from matplotlib import pyplot as plt

import desolver as de
import desolver.backend as D

D.set_float_fmt('float64')
```

```
Using numpy backend
```

Specifying the Dynamical System

Now let's specify the right hand side of our dynamical system. It should be

$$\ddot{x} + \delta\dot{x} + \alpha x + \beta x^3 = \gamma\cos(\omega t)$$

But desolver only works with first order differential equations, thus we must cast this into a first order system before we can solve it. Thus we obtain the following system

$$\begin{array}{l}
\frac{\mathrm{d}x}{\mathrm{dt}} = v_x \\
\frac{\mathrm{d}v_x}{\mathrm{dt}} = -\delta v_x - \alpha x - \beta x^3 + \gamma\cos(\omega t)
\end{array}$$

[2]:
```
@de.rhs_prettifier(
    equ_repr="[vx, -*vx - α*x - β*x**3 + γ*cos(ω*t)]",
    md_repr=r"""
$$
\begin{array}{l}
\frac{\mathrm{d}x}{\mathrm{dt}} = v_x \\
\frac{\mathrm{d}v_x}{\mathrm{dt}} = -\delta v_x - \alpha x - \beta x^3 +␣
↪\gamma\cos(\omega t)
\end{array}
$$
"""
)
def rhs(t, state, delta, alpha, beta, gamma, omega, **kwargs):
    x,vx = state
    return D.stack([
        vx,
        -delta*vx - alpha*x - beta*x**3 + gamma*D.cos(omega*t)
    ])
```

[3]:
```
print(rhs)
display(rhs)
```

```
[vx, -*vx - *x - *x**3 + *cos(*t)]
```

$$\begin{array}{l}
\frac{\mathrm{d}x}{\mathrm{dt}} = v_x \\
\frac{\mathrm{d}v_x}{\mathrm{dt}} = -\delta v_x - \alpha x - \beta x^3 + \gamma\cos(\omega t)
\end{array}$$

Let's specify the initial conditions as well

[4]:
```
y_init = D.array([1., 0.])
```

And now we're ready to integrate!

The Numerical Integration

We will use the same constants from Wikipedia as our constants where the forcing amplitude increases and all the other parameters stay constants.

[5]:
```
#Let's define the fixed constants

constants = dict(
    delta = 0.3,
```

(continues on next page)

(continued from previous page)

```
    omega = 1.2,
    alpha = -1.0,
    beta  = 1.0
)

# The period of the system
T = 2*D.pi / constants['omega']

# Initial and Final integration times
t0 = 0.0
tf = 40 * T
```

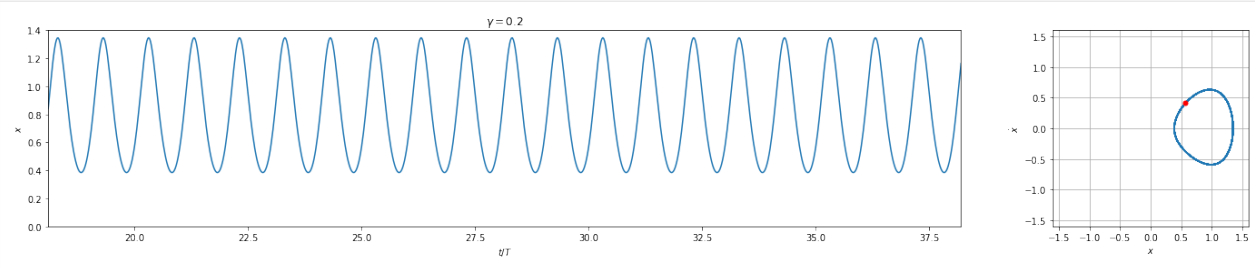
[6]:
```
a = de.OdeSystem(rhs, y0=y_init, dense_output=True, t=(t0, tf), dt=0.01, rtol=1e-12,␣
↪atol=1e-12, constants={**constants})
a.method = "RK87"
```

[7]:
```
a.reset()
a.constants['gamma'] = 0.2
a.integrate()
```

Plotting the State and Phase Portrait

[8]:
```
# Times to evaluate the system at
eval_times = D.linspace(18.1, 38.2, 1000)*T
```

[9]:
```
from matplotlib import gridspec

fig = plt.figure(figsize=(20, 4))

gs  = gridspec.GridSpec(1, 2, width_ratios=[3, 1])
ax0 = fig.add_subplot(gs[0])
ax1 = fig.add_subplot(gs[1])
ax1.set_aspect(1)

ax0.plot(eval_times/T, a.sol(eval_times)[:, 0])
ax0.set_xlim(18.1, 38.2)
ax0.set_ylim(0, 1.4)
ax0.set_xlabel(r"$t/T$")
ax0.set_ylabel(r"$x$")
ax0.set_title(r"$\gamma={}$".format(a.constants['gamma']))

ax1.plot(a[18*T:].y[:, 0], a[18*T:].y[:, 1])
ax1.scatter(a[18*T:].y[-1:, 0], a[18*T:].y[-1:, 1], c='red', s=25, zorder=10)
ax1.set_xlim(-1.6, 1.6)
ax1.set_ylim(-1.6, 1.6)
ax1.set_xlabel(r"$x$")
ax1.set_ylabel(r"$\dot x$")
ax1.grid(which='major')
plt.tight_layout()
```



We can see that this plot looks near identical to this plot.

$$\gamma = 0.20$$

Integrating for Different Values of Gamma

Now let's try to recreate the other plots with the varying gamma values.

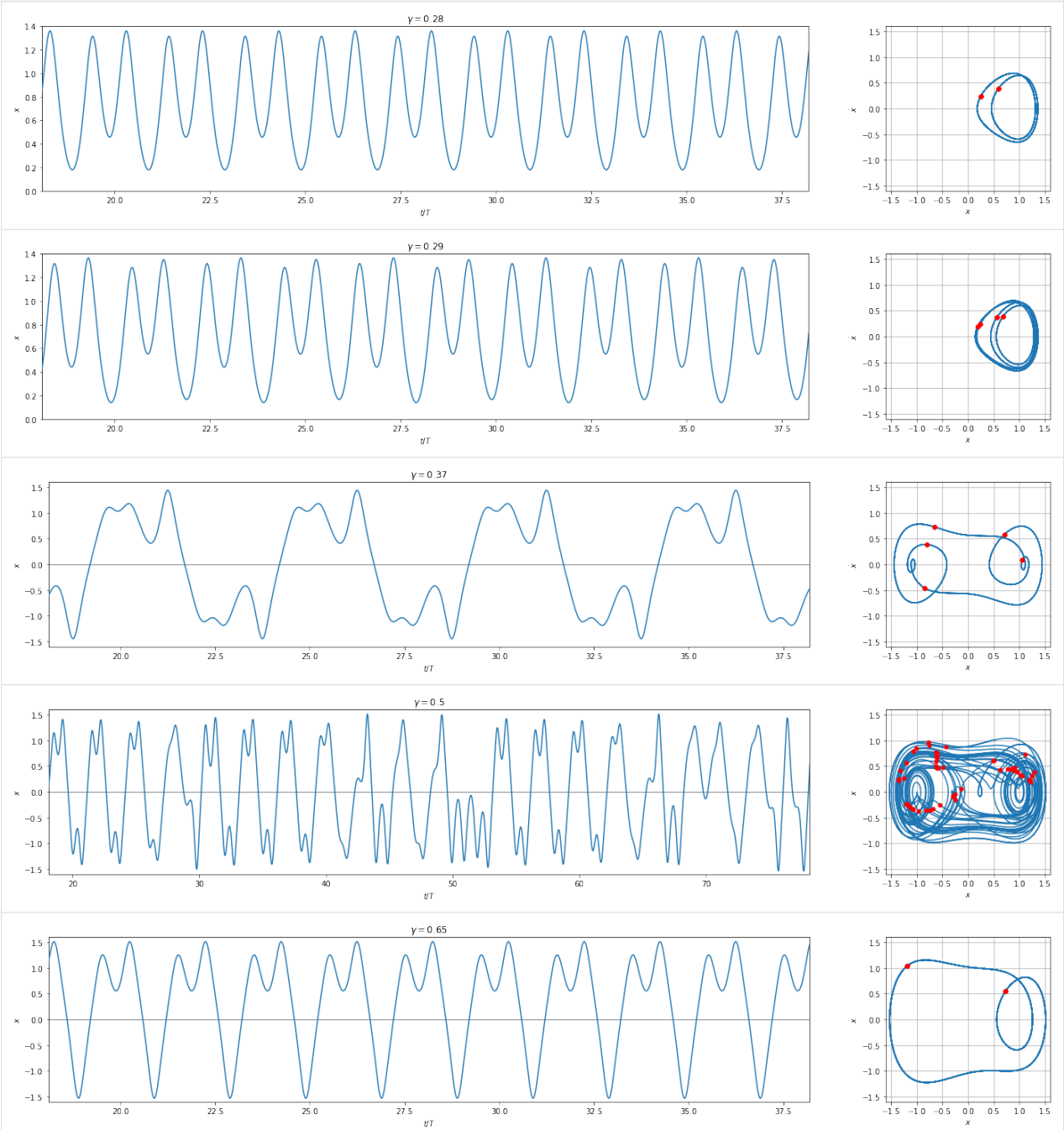
[10]:
```
gamma_values = [0.28, 0.29, 0.37, 0.5, 0.65]
integration_results = []

for gamma in gamma_values:
    a.reset()
    a.constants['gamma'] = gamma

    if gamma == 0.5:
        a.tf = 80.*T
        eval_times = D.linspace(18.1, 78.2, 3000)*T
        integer_period_multiples = D.arange(19, 78)*T
    else:
        a.tf = 40.*T
        eval_times = D.linspace(18.1, 38.2, 1000)*T
        integer_period_multiples = D.arange(19, 38)*T

    a.integrate()
    integration_results.append(((eval_times, a.sol(eval_times)), a.sol(integer_period_
↪multiples)))
```

[11]:
```
for gamma, ((state_times, states), period_states) in zip(gamma_values, integration_
↪results):
    fig = plt.figure(figsize=(20, 4))

    gs  = gridspec.GridSpec(1, 2, width_ratios=[3, 1])
    ax0 = fig.add_subplot(gs[0])
    ax1 = fig.add_subplot(gs[1])
    ax1.set_aspect(1)

    ax0.plot(state_times/T, states[:, 0], zorder=9)
    ax0.set_xlim(state_times[0]/T, state_times[-1]/T)
    if gamma < 0.37:
        ax0.set_ylim(0, 1.4)
    else:
        ax0.set_ylim(-1.6, 1.6)
        ax0.axhline(0, color='k', linewidth=0.5)
    ax0.set_xlabel(r"$t/T$")
    ax0.set_ylabel(r"$x$")
```

(continues on next page)

(continued from previous page)

```
    ax0.set_title(r"$\gamma={}$".format(gamma))

    ax1.plot(states[:, 0], states[:, 1])
    ax1.scatter(period_states[:, 0], period_states[:, 1], c='red', s=25, zorder=10)
    ax1.set_xlim(-1.6, 1.6)
    ax1.set_ylim(-1.6, 1.6)
    ax1.set_xlabel(r"$x$")
    ax1.set_ylabel(r"$\dot x$")
    ax1.grid(which='major')
    plt.tight_layout()
```



And we see that these plots replicate those on Wikipedia very nicely except for the chaotic one which is highly sensitive to the numerical integrator and tolerances.

(link to image) $\gamma = 0.28$

(link to image) $\gamma = 0.29$

(link to image) $\gamma = 0.37$

(link to image) $\gamma = 0.50$

(link to image) $\gamma = 0.65$

6.4.3 N-Body Gravitationally Interacting System

Let's try doing something more complicated: N-body dynamics. As the name implies, we have N interacting bodies where the interactions are mediated by some forces.

For this notebook, we'll look at gravitational interactions. It should be mentioned that N-body interactions are generally slow to compute because every body interacts with every other body and thus you have to compute the force $N(N-1)$, $\frac{N(N-1)}{2}$ times if the force is symmetric as in our case, which scales really poorly. There are methods to overcome this, but are not in the scope of this notebook as we'll be looking at $N \sim 10$.

First we import the libraries we'll need. I import all the matplotlib machinery using the magic command `%matplotlib`, but this is only for notebook/ipython environments.

Then I import `desolver` and the desolver backend as well (this will be useful for specifying our problem), and set the default datatype to `float64`.

[1]:
```
%matplotlib inline
from matplotlib import pyplot as plt

import desolver as de
import desolver.backend as D

D.set_float_fmt('float64')
```

```
Using numpy backend
```

Specifying the Dynamical System

Writing out the dynamical equations will take a bit more work, but we'll start from Newton's Law of Gravitation and build from there.

So, Newton's Law of Gravitation states the gravitational force exerted by one body on another is

$$\vec{F}_{ij} = G\frac{m_i m_j}{|\vec{r}_j - \vec{r}_i|^3}(\vec{r}_j - \vec{r}_i)$$

where G is the gravitational constant, m_i and m_j are the masses of the two bodies, and $\vec{r}_i$ and $\vec{r}_j$ are the position vectors of the two bodies.

Note that this is symmetric in terms of magnitude and merely the direction is flipped when we swap i and j, i.e. $\vec{F}_{ij} = -\vec{F}_{ji}$. This is where the factor of $\frac{1}{2}$ comes from the in the number of computations required.

This is not sufficient to compute the actual dynamics, we must find the force on a single body from all the other bodies. To do this, we simply add the forces of every other body in the simulation thus the force is

$$\vec{F}_i = \sum_{j \neq i} F_{ij}$$

You'll notice that we sum over $i \neq j$ thus there are $(N-1)$ F_{ij} terms to add. This is where the factor of $(N-1)$ in the number of computations comes from, and since we do it for every single body, we do this computation N times thus the factor of N is explained as well.

This is all well and good, but how do we write this? Well, `desolver` makes that part much simpler. Unlike solve_ivp, my equation representation is not required to be a single vector at any stage, thus I can easily have an array of shape `(N, 6)` to represent the state vectors of each body. This will also allow me to easily take advantage of `numpy` broadcasting semantics.

[2]:
```
def Fij(ri, rj, G):
    rel_r = rj - ri
    return G*(1/D.norm(rel_r, ord=2)**3)*rel_r

def rhs(t, state, masses, G):
    total_acc = D.zeros_like(state)

    for idx, (ri, mi) in enumerate(zip(state, masses)):
        for jdx, (rj, mj) in enumerate(zip(state[idx+1:], masses[idx+1:])):
            partial_force = Fij(ri[:3], rj[:3], G)
            total_acc[idx, 3:]       += partial_force * mj
            total_acc[idx+jdx+1, 3:] -= partial_force * mi

    total_acc[:, :3] = state[:, 3:]

    return total_acc
```

NOTE: The line `total_acc[:, :3] -= (D.sum(total_acc[:, :3]*masses[:, None], axis=0) / D.sum(masses))` is useful when we want to look at how the objects behave relative to the centre of mass. Generally this would be relative to the Sun if the other bodies are the planets in our Solar System given that most of the mass in our Solar System is concentrated at the Sun.

Hénon Units

Now we cannot use the equations in this form because they encompass a very large scale of values ranging from 10^{-11} for the gravitational constant to 10^{30} for the mass of the sun. Thus we will non-dimensionalise the system by measuring all the lengths by $1AU = 149597871km$, the masses by $M_{\odot} = 1.989 \times 10^{30}kg$, and the time in $1yr$.

In these units, the gravitational constant becomes $G = 4\pi^2$. Using the constants defined in the next cell, units in SI can be converted to units in our system.

[3]:
```
Msun = 1.98847*10**30    ## Mass of the Sun, kg
AU   = 149597871e3       ## 1 Astronomical Unit, m
year = 365.25*24*3600    ## 1 year, s
G    = 4*D.pi**2         ## in solar masses, AU, years
V    = D.sqrt(G)         ## Speed scale corresponding to the orbital speed required␣
↪for a circular orbit at 1AU with a period of 1yr
```

I've added 3 massive bodies at the ends of a scalene triangle

[4]:
```
initial_state = D.array([
    [0.0,   0.0,                1.0,  0.0,                  -1.0,    0.0],
    [1.0,   0.0,                0.0,  0.0,                   1.0,    0.0],
    [0.25,  0.9682458365518543, 0.0,  0.9682458365518543*0, -0.25*0, 0.0],
    [-0.5, -0.8660254037844386, 0.0, -0.8660254037844386*0,  0.5*0,  0.0],
])
```

(continues on next page)

(continued from previous page)

```
masses = D.array([
    1,
    1,
    1,
    1,
])

rhs(0.0, initial_state, masses, G)
```

[4]:
```
array([[  0.        ,  -1.        ,   0.        ,  10.4682963 ,
          1.42676504, -41.8731852 ],
       [  0.        ,   1.        ,   0.        , -41.47116241,
         14.22721674,  13.9577284 ],
       [  0.        ,  -0.        ,   0.        ,   8.82285763,
        -43.62661768,  13.9577284 ],
       [ -0.        ,   0.        ,   0.        ,  22.18000848,
         27.97263591,  13.9577284 ]])
```

The Numerical Integration

We will use the 14th order Runge-Kutta integrator to integrate our system with tolerances of 10^{-14} which should give fairly accurate results despite the very complicated trajectories of each body.

Although symplectic integrators are great for conserving the energy, we prefer an adaptive integrator due to the fact that the bodies have close encounters which require much smaller steps to resolve. If we were to use small step sizes at all times, we'd be wasting computation whenever the bodies are far from each other. This leads us to pick an adaptive Runge-Kutta method.

[5]:
```
a = de.OdeSystem(rhs, y0=initial_state, dense_output=True, t=(0, 2.0), dt=0.00001,␣
↪rtol=1e-14, atol=1e-14, constants=dict(G=G, masses=masses))
a.method = "RK1412"
```

[6]:
```
a.integrate()
```

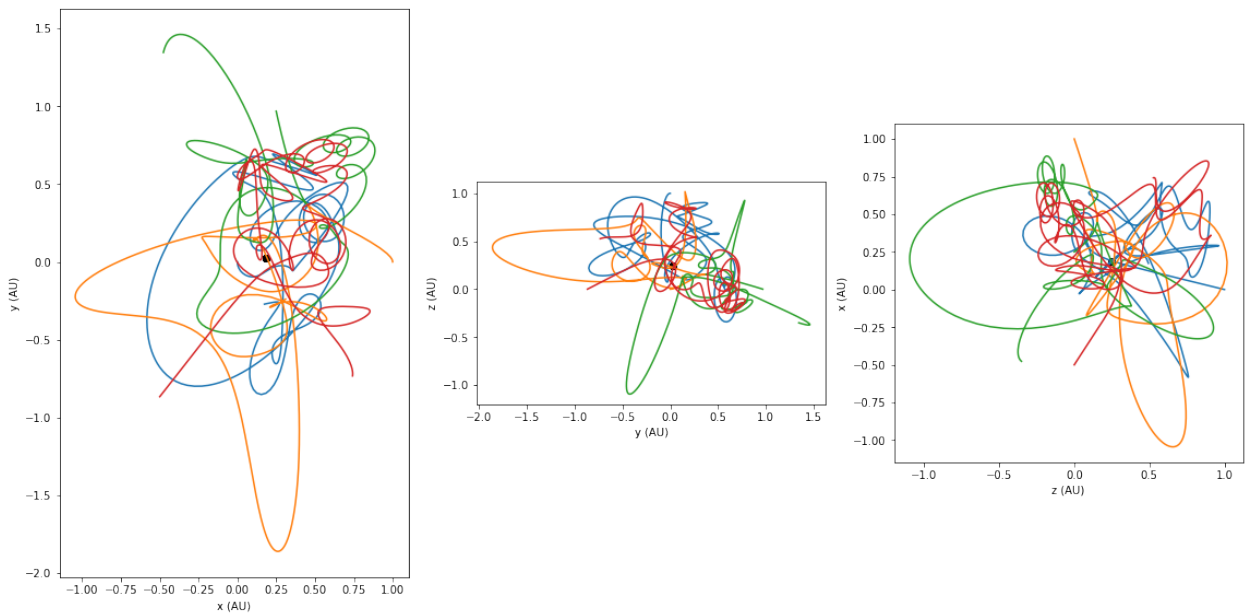
[7]:
```
fig = plt.figure(figsize=(16,16))

com_motion = D.sum(a.y[:, :, :] * masses[None, :, None], axis=1) / D.sum(masses)

fig = plt.figure(figsize=(16,16))
ax1 = fig.add_subplot(131, aspect=1)
ax2 = fig.add_subplot(132, aspect=1)
ax3 = fig.add_subplot(133, aspect=1)

ax1.set_xlabel("x (AU)")
ax1.set_ylabel("y (AU)")
ax2.set_xlabel("y (AU)")
ax2.set_ylabel("z (AU)")
ax3.set_xlabel("z (AU)")
ax3.set_ylabel("x (AU)")

for i in range(a.y.shape[1]):
    ax1.plot(a.y[:, i, 0], a.y[:, i, 1], color=f"C{i}")
    ax2.plot(a.y[:, i, 1], a.y[:, i, 2], color=f"C{i}")
    ax3.plot(a.y[:, i, 2], a.y[:, i, 0], color=f"C{i}")
```

(continues on next page)

(continued from previous page)

```
ax1.scatter(com_motion[:, 0], com_motion[:, 1], color='k')
ax2.scatter(com_motion[:, 1], com_motion[:, 2], color='k')
ax3.scatter(com_motion[:, 2], com_motion[:, 0], color='k')

plt.tight_layout()
```

```
<Figure size 1152x1152 with 0 Axes>
```



Close Encounters and Event Detection

Suppose we were interested in finding when two bodies are near each other, how would we do it? We could integrate the system, compare the distances of each body with one another and whenever they were below some threshold we can mark that time. But this is inherently inexact because maybe the actual closest point occurs between two timesteps and our adaptive integrator, being extra intelligent, did not require stepping through that exact point to find the closest encounter.

So what can we do? We can use **Event Detection**! Event detection is where we find when some event occurs during a numerical integration and localise on that event.

We need to first create a function that will tell us when two objects are nearby. We'll define nearby as whenever the two bodies are within half of each other's Hill sphere. The Hill radius is computed as $a\sqrt[3]{\frac{m}{3M_{total}}}$ where we assume that the eccentricity is unimportant and a will be the distance from the centre of mass.

[8]:
```
def close_encounter(t, state, masses, G):
    distances_between_bodies = []

    total_mass      = D.sum(masses)
    center_of_mass = D.sum(state[:, :3] * masses[:, None], axis=1) / total_mass

    com_distances  = D.norm(state[:, :3] - center_of_mass[:, None], axis=1)
    hill_radii     = com_distances * D.pow(masses/(3*total_mass), 1/3)

    for idx,ri in enumerate(state[:, :3]):
```

(continues on next page)

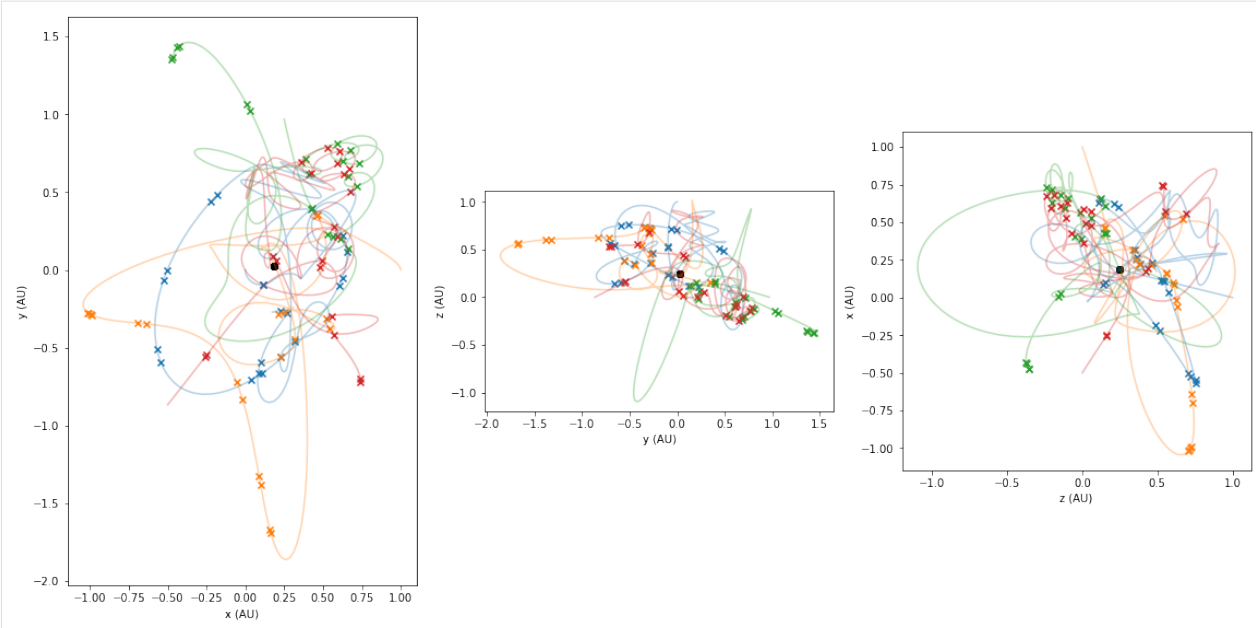
(continued from previous page)

```
        for jdx, rj in enumerate(state[idx+1:, :3]):
            distances_between_bodies.append(D.norm(ri - rj) - D.min([hill_radii[idx],␣
↪hill_radii[jdx]])/2.0)

    return D.min(distances_between_bodies)
```

[9]:
```
a.reset()
a.integrate(events=close_encounter)
```

[10]:
```
fig = plt.figure(figsize=(16,16))

com_motion = D.sum(a.y[:, :, :] * masses[None, :, None], axis=1) / D.sum(masses)

fig = plt.figure(figsize=(16,16))
ax1 = fig.add_subplot(131, aspect=1)
ax2 = fig.add_subplot(132, aspect=1)
ax3 = fig.add_subplot(133, aspect=1)

ax1.set_xlabel("x (AU)")
ax1.set_ylabel("y (AU)")
ax2.set_xlabel("y (AU)")
ax2.set_ylabel("z (AU)")
ax3.set_xlabel("z (AU)")
ax3.set_ylabel("x (AU)")

for i in range(a.y.shape[1]):
    ax1.plot(a.y[:, i, 0], a.y[:, i, 1], color=f"C{i}", alpha=0.33)
    ax2.plot(a.y[:, i, 1], a.y[:, i, 2], color=f"C{i}", alpha=0.33)
    ax3.plot(a.y[:, i, 2], a.y[:, i, 0], color=f"C{i}", alpha=0.33)
    for j in a.events:
        ax1.scatter(j.y[i, 0], j.y[i, 1], c=f"C{i}", marker='x', alpha=1.0)
        ax2.scatter(j.y[i, 1], j.y[i, 2], c=f"C{i}", marker='x', alpha=1.0)
        ax3.scatter(j.y[i, 2], j.y[i, 0], c=f"C{i}", marker='x', alpha=1.0)

ax1.scatter(com_motion[:, 0], com_motion[:, 1], color='k')
ax2.scatter(com_motion[:, 1], com_motion[:, 2], color='k')
ax3.scatter(com_motion[:, 2], com_motion[:, 0], color='k')

plt.tight_layout()
```

```
<Figure size 1152x1152 with 0 Axes>
```



We see that there are many close encounters and furthermore, the encounters are not restricted to any particular pairs of bodies, but sometimes happen with three bodies simultaneously. We will see this better in the next section where we look at an animation of the bodies.

An Animated View of the Bodies

The following code, although long, shows how to set up a matplotlib animation showing the bodies in all three planes and the close encounters they experience.

[11]:
```
from matplotlib import animation, rc

# set to location of ffmpeg to get animations working
# For Linux or Mac
# plt.rcParams['animation.ffmpeg_path'] = '/usr/bin/ffmpeg'

# For Windows
plt.rcParams['animation.ffmpeg_path'] = 'C:\\ProgramData\\chocolatey\\bin\\ffmpeg.exe'

from IPython.display import HTML
```

[12]:
```
%%capture

# This magic command prevents the creation of a static figure image so that we can␣
↪view the animation in the next cell
t          = a.t
all_states = a.y
planets    = [all_states[:, i, :] for i in range(all_states.shape[1])]
com_motion = D.sum(all_states * masses[None, :, None], axis=1) / D.sum(masses)

plt.ioff()

fig = plt.figure(figsize=(16,8))
ax1  = fig.add_subplot(131, aspect=1)
```

(continues on next page)

(continued from previous page)

```
ax2  = fig.add_subplot(132, aspect=1)
ax3  = fig.add_subplot(133, aspect=1)

ax1.set_xlabel("x (AU)")
ax1.set_ylabel("y (AU)")
ax2.set_xlabel("y (AU)")
ax2.set_ylabel("z (AU)")
ax3.set_xlabel("z (AU)")
ax3.set_ylabel("x (AU)")

xlims = D.abs(a.y[:, :, 0]).max()
ylims = D.abs(a.y[:, :, 1]).max()
zlims = D.abs(a.y[:, :, 2]).max()

ax1.set_xlim(-xlims-0.25, xlims+0.25)
ax2.set_xlim(-ylims-0.25, ylims+0.25)
ax3.set_xlim(-zlims-0.25, zlims+0.25)

ax1.set_ylim(-ylims-0.25, ylims+0.25)
ax2.set_ylim(-zlims-0.25, zlims+0.25)
ax3.set_ylim(-xlims-0.25, xlims+0.25)

planets_pos_xy = []
planets_pos_yz = []
planets_pos_zx = []

planets_xy     = []
planets_yz     = []
planets_zx     = []

com_xy,        = ax1.plot([], [], color='k', linestyle='', marker='o', markersize=5.0,
↪ zorder=10)
com_yz,        = ax2.plot([], [], color='k', linestyle='', marker='o', markersize=5.0,
↪ zorder=10)
com_zx,        = ax3.plot([], [], color='k', linestyle='', marker='o', markersize=5.0,
↪ zorder=10)

event_counter = 0

close_encounter_xy = []
close_encounter_yz = []
close_encounter_zx = []

for i in range(len(planets)):
    close_encounter_xy.append(ax1.plot([], [], color=f"k", marker='x', markersize=3.0,
↪ linestyle='', zorder=9)[0])
    close_encounter_yz.append(ax2.plot([], [], color=f"k", marker='x', markersize=3.0,
↪ linestyle='', zorder=9)[0])
    close_encounter_zx.append(ax3.plot([], [], color=f"k", marker='x', markersize=3.0,
↪ linestyle='', zorder=9)[0])

for i in range(a.y.shape[1]):
    planets_xy.append(ax1.plot([], [], color=f"C{i}", zorder=8)[0])
    planets_yz.append(ax2.plot([], [], color=f"C{i}", zorder=8)[0])
    planets_zx.append(ax3.plot([], [], color=f"C{i}", zorder=8)[0])
    planets_pos_xy.append(ax1.plot([], [], color=f"C{i}", linestyle='', marker='.',␣
↪zorder=8)[0])
```

(continues on next page)

(continued from previous page)

```
    planets_pos_yz.append(ax2.plot([], [], color=f"C{i}", linestyle='', marker='.',␣
↪zorder=8)[0])
    planets_pos_zx.append(ax3.plot([], [], color=f"C{i}", linestyle='', marker='.',␣
↪zorder=8)[0])

def init():
    global event_counter
    for i in range(len(planets)):
        planets_xy[i].set_data([], [])
        planets_yz[i].set_data([], [])
        planets_zx[i].set_data([], [])
        planets_pos_xy[i].set_data([], [])
        planets_pos_yz[i].set_data([], [])
        planets_pos_zx[i].set_data([], [])

    com_xy.set_data([], [])
    com_yz.set_data([], [])
    com_zx.set_data([], [])

    for i in range(len(planets)):
        close_encounter_xy[i].set_data(a.events[event_counter].y[i, 0], a.
↪events[event_counter].y[i, 1])
        close_encounter_yz[i].set_data(a.events[event_counter].y[i, 1], a.
↪events[event_counter].y[i, 2])
        close_encounter_zx[i].set_data(a.events[event_counter].y[i, 2], a.
↪events[event_counter].y[i, 0])

    return tuple(planets_xy + planets_yz + planets_zx + planets_pos_xy + planets_pos_
↪yz + planets_pos_zx + [com_xy, com_yz, com_zx] + [close_encounter_xy, close_
↪encounter_yz, close_encounter_zx])

def animate(frame_num):
    global event_counter
    for i in range(len(planets)):
        planets_xy[i].set_data(planets[i][max(frame_num-5, 0):frame_num, 0],␣
↪planets[i][max(frame_num-5, 0):frame_num, 1])
        planets_yz[i].set_data(planets[i][max(frame_num-5, 0):frame_num, 1],␣
↪planets[i][max(frame_num-5, 0):frame_num, 2])
        planets_zx[i].set_data(planets[i][max(frame_num-5, 0):frame_num, 2],␣
↪planets[i][max(frame_num-5, 0):frame_num, 0])
        planets_pos_xy[i].set_data(planets[i][frame_num:frame_num+1, 0],␣
↪planets[i][frame_num:frame_num+1, 1])
        planets_pos_yz[i].set_data(planets[i][frame_num:frame_num+1, 1],␣
↪planets[i][frame_num:frame_num+1, 2])
        planets_pos_zx[i].set_data(planets[i][frame_num:frame_num+1, 2],␣
↪planets[i][frame_num:frame_num+1, 0])

    com_xy.set_data(com_motion[frame_num:frame_num+1, 0], com_motion[frame_num:frame_
↪num+1, 1])
    com_yz.set_data(com_motion[frame_num:frame_num+1, 1], com_motion[frame_num:frame_
↪num+1, 2])
    com_zx.set_data(com_motion[frame_num:frame_num+1, 2], com_motion[frame_num:frame_
↪num+1, 0])

    if t[frame_num] >= a.events[event_counter].t and event_counter + 1 < len(a.
↪events):
        event_counter += 1
```

(continues on next page)

(continued from previous page)

```
        for i in range(len(planets)):
            close_encounter_xy[i].set_data(a.events[event_counter].y[i, 0], a.
↪events[event_counter].y[i, 1])
            close_encounter_yz[i].set_data(a.events[event_counter].y[i, 1], a.
↪events[event_counter].y[i, 2])
            close_encounter_zx[i].set_data(a.events[event_counter].y[i, 2], a.
↪events[event_counter].y[i, 0])

    return tuple(planets_xy + planets_yz + planets_zx + planets_pos_xy + planets_pos_
↪yz + planets_pos_zx + [com_xy, com_yz, com_zx] + [close_encounter_xy, close_
↪encounter_yz, close_encounter_zx])

ani = animation.FuncAnimation(fig, animate, list(range(1, len(t))),
                              interval=1500./60., blit=False, init_func=init)

rc('animation', html='html5')

# Uncomment to save an mp4 video of the animation
# ani.save('Nbodies.mp4', fps=60)
```

Here we see that the animation slows down whenever the bodies come close to each other and this is due to the adaptive timestepping of the numerical integration which takes more steps whenever there is a close encounter. Each "**x**" marks the point of all the bodies whenever there is a close encounter.

Additionally, it's interesting to see that the center of mass (the large black dot in the center) does not move at all. Since we initialised the system with zero momentum, this is to be expected given that we are not doing anything to violate momentum conservation, but it is good to see that our numerical integration also respects this behaviour.

[13]:
```
display(ani)
```

```
<matplotlib.animation.FuncAnimation at 0x24084c63cc8>
```

6.5 Using pyaudi

6.5.1 Differential Intelligence

(original by Dario Izzo - extended by Ekin Ozturk)

In this notebook we show the use of `desolver` for numerical integration following the notebook example here differential intelligence.

Importing Stuff

[1]:
```
import pyaudi
```

[2]:
```
%matplotlib inline
from matplotlib import pyplot as plt

import os
import numpy as np

os.environ['DES_BACKEND'] = 'numpy'
```

(continues on next page)

(continued from previous page)

```
import desolver as de
import desolver.backend as D

D.set_float_fmt('gdual_double')
```

```
PyAudi backend is available.
Using numpy backend
```

Controller representation and "simulator"

Take as an example the task of learning a robotic controller. In neuro evolution (or Evolutionary Robotics), the controller is typically represented by a neural network, but for the purpose of explaining this new learning concept we will use a polinomial representation for the controller. Later, changing the controller into a NN with weights as parameters will not change the essence of what is done here.

[3]:
```
# Definition of the controller in terms of some weights parameters
def u(state, weights):
    x,v = state
    a,b,c,e,f,g = weights
    return a + b*x + c*v + e*x*v + f*x**2 + g*v**2
```

[4]:
```
# Definition of the equation of motion (our physics simulator propagating the system␣
↪to its next state)
def eom(state, weights):
    x,v = state
    dx = v
    dv = u(state, weights)
    return (dx, dv)
```

Numerical Integration - Runge-Kutta 45 Cash-Karp Method

In Evolutionary Robotics, Euler propagators are commonly used, but we would like to use a higher order integration scheme that is adaptive in order to minimise computation, and increase the accuracy and precision of the results.

[5]:
```
weights = D.array([D.gdual_double(0.05*(np.random.uniform()-0.5), _, 7) for _ in
↪"abcefg"])
x = [D.gdual_double(2*(np.random.uniform()-0.5))]
v = [D.gdual_double(2*(np.random.uniform()-0.5))]
y0 = D.array(x + v, dtype=D.gdual_double)
```

[6]:
```
def rhs(t, state, weights, **kwargs):
    return D.array(eom(state, weights))
```

We integrate the system using the Runge-Kutta-Cash-Karp scheme as the numerical integration system with a dense output computed using a piecewise C1 Hermite interpolating spline.

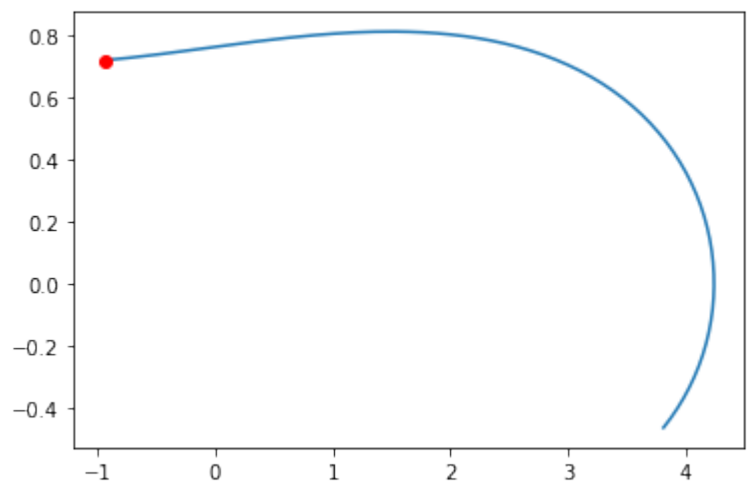
This particular interpolator is used as it satisfies not only the state boundary conditions, but also the gradients and is well suited for approximating the solution continuously up to first order in time.

Note that the entire integration is done using gduals and thus the solution and the interpolating spline stored in the OdeSystem instance, `pyaudi_integration`, contains all the information about how the state reached by our robot changes when we change the control weights.

[7]:
```
pyaudi_integration = de.OdeSystem(rhs, y0=y0, dense_output=False, t=(0, 10.), dt=0.1,␣
↪rtol=1e-12, atol=1e-12, constants=dict(weights=weights))

pyaudi_integration.set_method("RK45")
pyaudi_integration.integrate(eta=True)
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

[8]:
```
x,v = pyaudi_integration.y.T
plt.plot([it for it in x.astype(np.float64)],[it for it in v.astype(np.float64)])
plt.plot(x[0].constant_cf, v[0].constant_cf, 'ro')
plt.show()
```



[9]:
```
xf, vf = x[-1], v[-1]
```

[10]:
```
print("initial xf: {}".format(xf.constant_cf))
print("initial vf: {}".format(vf.constant_cf))
```

```
initial xf: 3.8065257635631173
initial vf: -0.4637962920728024
```

Studying the effects of the weights on the behavior

We have represented all the robot behavior (x, v) as a polynomial function of the weights. So we now know what happens to the behaviour if we change the weights!! Lets see … we only consider the final state, but the same can be done for all states before.

[11]:
```
dweights = dict({'da': -0.002, 'db': 0.003, 'dc': -0.02, 'de': 0.03, 'df': 0.02, 'dg':
↪ -0.01})
#Lets predict the new final position of our 'robot' if we change his controller as␣
↪defined above
print("new xf: {}".format(xf.evaluate(dweights)))
print("new vf: {}".format(vf.evaluate(dweights)))
```

```
new xf: 11.843378895251684
new vf: 3.5577687770021136
```

Check that we learned the correct map

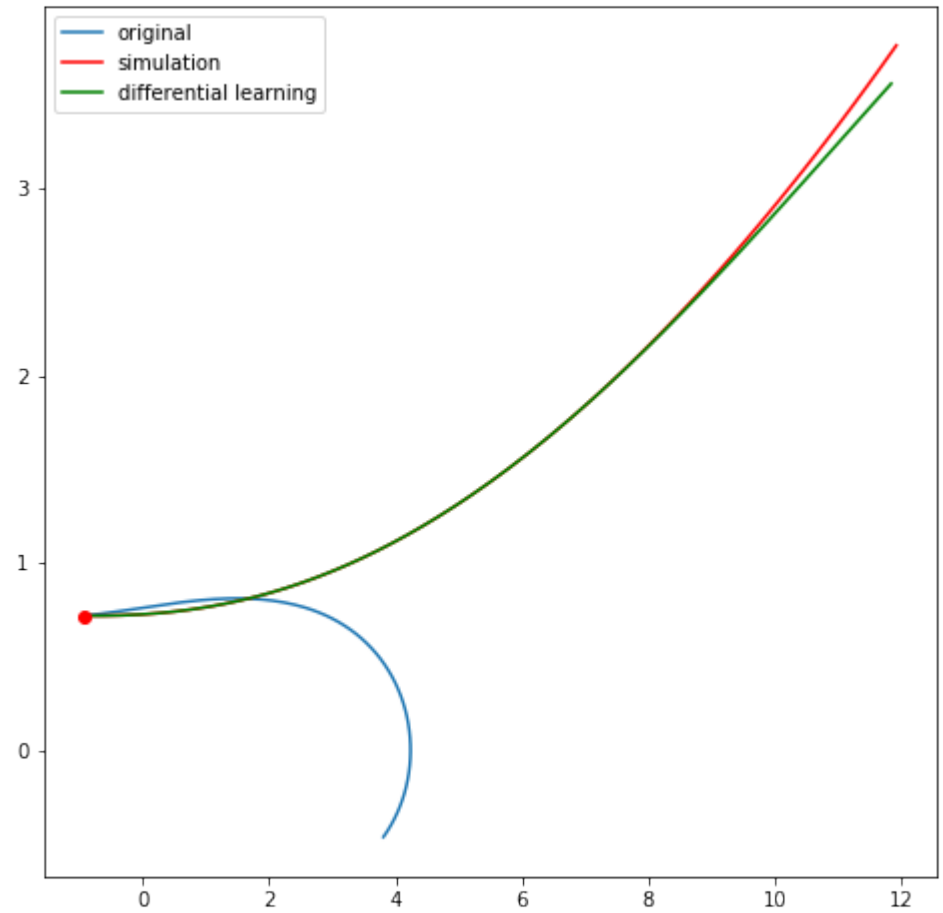
We now simulate again our behavior using the new weights to see where we end up to check if the prediction made after our differential learning is correct.

[12]:
```
new_weights = np.array([it + dweights['d' + it.symbol_set[0]] for it in weights])
```

[13]:
```
pyaudi_integration2 = de.OdeSystem(rhs, y0=y0, dense_output=True, t=(0, 10.), dt=0.01,
↪ rtol=1e-12, atol=1e-12, constants=dict(weights=new_weights))

pyaudi_integration2.set_method("RK45")
pyaudi_integration2.integrate(eta=True)
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

[14]:
```
plt.figure(figsize=(8,8))
x2, v2 = pyaudi_integration2.y.T
plt.plot([it.constant_cf for it in x],[it.constant_cf for it in v],label='original')
plt.plot([it.constant_cf for it in x2],[it.constant_cf for it in v2],'r',label=
↪'simulation')
plt.plot([it.evaluate(dweights) for it in x],[it.evaluate(dweights) for it in v],'g',
↪label='differential learning')
plt.plot(x[0].constant_cf, v[0].constant_cf, 'ro')
plt.legend(loc=2)
plt.show()
```



[15]:
```
print("Differential learning xf: \t{}".format(x[-1].evaluate(dweights)))
print("Real xf: \t\t\t{}".format(x2[-1].constant_cf))
print("Mean Absolute Difference xf:\t{}".format(D.max(D.abs(D.to_float(x2[-1]) - D.to_
↪float(x[-1].evaluate(dweights))))))
print()
print("Differential learning vf: \t{}".format(v[-1].evaluate(dweights)))
print("Real vf: \t\t\t{}".format(v2[-1].constant_cf))
print("Mean Absolute Difference vf:\t{}".format(D.max(D.abs(D.to_float(v2[-1]) - D.to_
↪float(v[-1].evaluate(dweights))))))
```

```
Differential learning xf:       11.843378895251684
Real xf:                        11.924939296996673
Mean Absolute Difference xf:    0.08156040174498891

Differential learning vf:       3.5577687770021136
Real vf:                        3.7620227657073806
Mean Absolute Difference vf:    0.204253988705267
```

6.5.2 Differential Intelligence - Vectorised Duals

(original by Dario Izzo - extended by Ekin Ozturk)

In this notebook we show the use of `desolver` and vectorised gduals for the numerical integration of multiple initial conditions following the notebook example here differential intelligence.

Importing Stuff

[1]:
```
%matplotlib inline
from matplotlib import pyplot as plt

import os
import numpy as np

os.environ['DES_BACKEND'] = 'numpy'
import desolver as de
import desolver.backend as D

D.set_float_fmt('gdual_vdouble')
```

```
PyAudi backend is available.
Using numpy backend
```

Controller representation and "simulator"

Take as an example the task of learning a robotic controller. In neuro evolution (or Evolutionary Robotics), the controller is typically represented by a neural network, but for the purpose of explaining this new learning concept we will use a polinomial representation for the controller. Later, changing the controller into a NN with weights as parameters will not change the essence of what is done here.

[2]:
```
# Definition of the controller in terms of some weights parameters
def u(state, weights):
    x,v = state
    a,b,c,e,f,g = weights
    return a + b*x + c*v + e*x*v + f*x**2 + g*v**2
```

[3]:
```
# Definition of the equation of motion (our physics simulator propagating the system␣
↪to its next state)
def eom(state, weights):
    x,v = state
    dx = v
    dv = u(state, weights)
    return (dx, dv)
```

Numerical Integration - Runge-Kutta 45 Cash-Karp Method

In Evolutionary Robotics, Euler propagators are commonly used, but we would like to use a higher order integration scheme that is adaptive in order to minimise computation, and increase the accuracy and precision of the results.

We are using `gdual_vdouble` in order to integrate multiple initial states simultaneously without any significant loss of computation time due to the very efficient vectorisation of `gdual_vdouble` computations.

[4]:
```
num_different_integrations = 16
weights = D.array([D.gdual_vdouble([0.2*(np.random.uniform()-0.5)]*num_different_
↪integrations, _, 4) for _ in "abcefg"])
x = [D.gdual_vdouble([2.*(np.random.uniform()-0.5) for i in range(num_different_
↪integrations)])]
v = [D.gdual_vdouble([2.*(np.random.uniform()-0.5) for i in range(num_different_
↪integrations)])]
y0 = D.array(x + v, dtype=D.gdual_vdouble)
```

[5]:
```
def rhs(t, state, weights, **kwargs):
    return D.array(eom(state, weights))
```

We integrate the system using the Runge-Kutta-Cash-Karp scheme as the numerical integration system with a dense output computed using a piecewise C1 Hermite interpolating spline.

This particular interpolator is used as it satisfies not only the state boundary conditions, but also the gradients and is well suited for approximating the solution continuously up to first order in time.

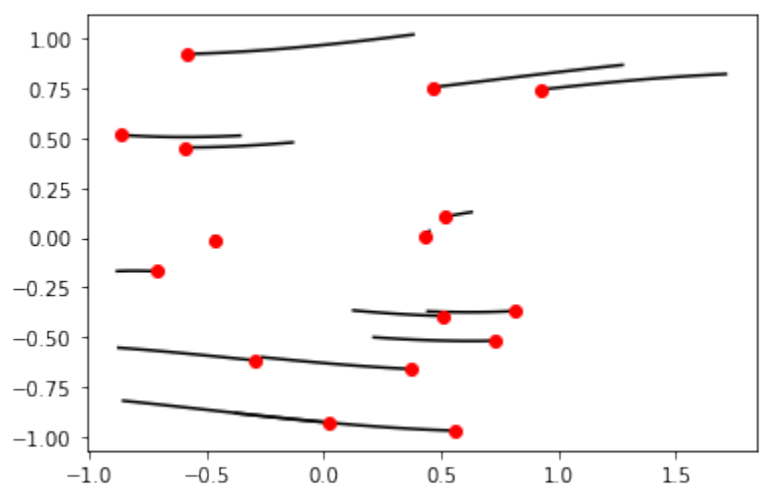
Note that the entire integration is done using gduals and thus the solution and the interpolating spline stored in the OdeSystem instance, `pyaudi_integration`, contains all the information about how the state reached by our robot changes when we change the control weights.

[6]:
```
# We restrict the integration time due to the fact that the controller
# is quadratic in the state and causes the state to grow very rapidly.

pyaudi_integration = de.OdeSystem(rhs, y0=y0, dense_output=True, t=(0, 3.), dt=0.1,␣
↪rtol=1e-12, atol=1e-12, constants=dict(weights=weights))

pyaudi_integration.set_method("RK45")
pyaudi_integration.integrate(eta=True)

x,v = pyaudi_integration.sol(D.linspace(0, 1, 20)).T
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

We numerically integrate 16 initial states and see the paths they follow based on the controller we have defined.

[7]:
```
for _x, _v in zip(D.to_float(x).T, D.to_float(v).T):
    plt.plot(_x, _v, 'k')
plt.plot(x[0].constant_cf, v[0].constant_cf, 'ro')
plt.show()
```



[8]:
```
xf, vf = x[-1], v[-1]
```

[9]:
```
print("initial xf: {}".format(xf.constant_cf))
print("initial vf: {}".format(vf.constant_cf))
```

```
initial xf: [-0.3557066099106508, 0.44343882132762114, -0.8737317576067235, -0.
↪2617441190669201, -0.8796623090368102, 0.3814675250018374, 0.4496686004618328, -0.
↪13184594443243638, -0.8531865179643819, 0.6296049480439482, 1.272489314100466, -0.
↪37367300287583055, 0.21369263660756216, -0.4685920682632495, 0.12662167090356946, 1.
↪7123448444138816]
initial vf: [0.5118610410272438, -0.3701236243745896, -0.5532466100960509, -0.
↪6008338241266351, -0.16869925311896916, 1.0191355842763685, 0.03313073890800292, 0.
↪47769851923104645, -0.8193041150168822, 0.12783245934453705, 0.8664324532172547, -0.
↪8819129622656402, -0.5007364437236071, 0.00378088557268934, -0.36570546484320804, 0.
↪8206755520959318]
```

Studying the effects of the weights on the behavior

We have represented all the robot behavior (x, v) as a polynomial function of the weights. So we now know what happens to the behaviour if we change the weights!! Lets see … we only consider the final state, but the same can be done for all states before.

Furthermore, we can compute this for all the different initial states that we integrated thus finding out how the final state of the robot changes for multiple initial conditions.

[10]:
```
dweights = dict({'da': -0.002, 'db': 0.003, 'dc': -0.02, 'de': 0.03, 'df': 0.02, 'dg':
↪ -0.01})
#Lets predict the new final position of our 'robot' if we change his controller as␣
↪defined above
print("new xf: {}".format(xf.evaluate(dweights)))
print("new vf: {}".format(vf.evaluate(dweights)))
```

```
new xf: [-0.3644798190145982, 0.447495385967943, -0.8648361515237677, -0.
↪2592939994695151, -0.8727114343058243, 0.3628173608150842, 0.4511668988597944, -0.
↪14014616132168833, -0.8452539766438874, 0.632267395802166, 1.2758792802868362, -0.
↪3718043764184628, 0.21623106958504587, -0.4679969881995721, 0.12859034018369273, 1.
↪7319060943023437]
new vf: [0.4936346648964618, -0.36314723947811245, -0.5327100009566508, -0.
↪5946732215257815, -0.15413715024294478, 0.9842045102773821, 0.0361826243667708, 0.
↪4608953840430959, -0.7987016506520095, 0.13376814463127482, 0.8819766632015573, -0.
↪8750423106238954, -0.49627765398605556, 0.004878367143968462, -0.36209906157800886,␣
↪0.8727946071182053]
```

Check that we learned the correct map

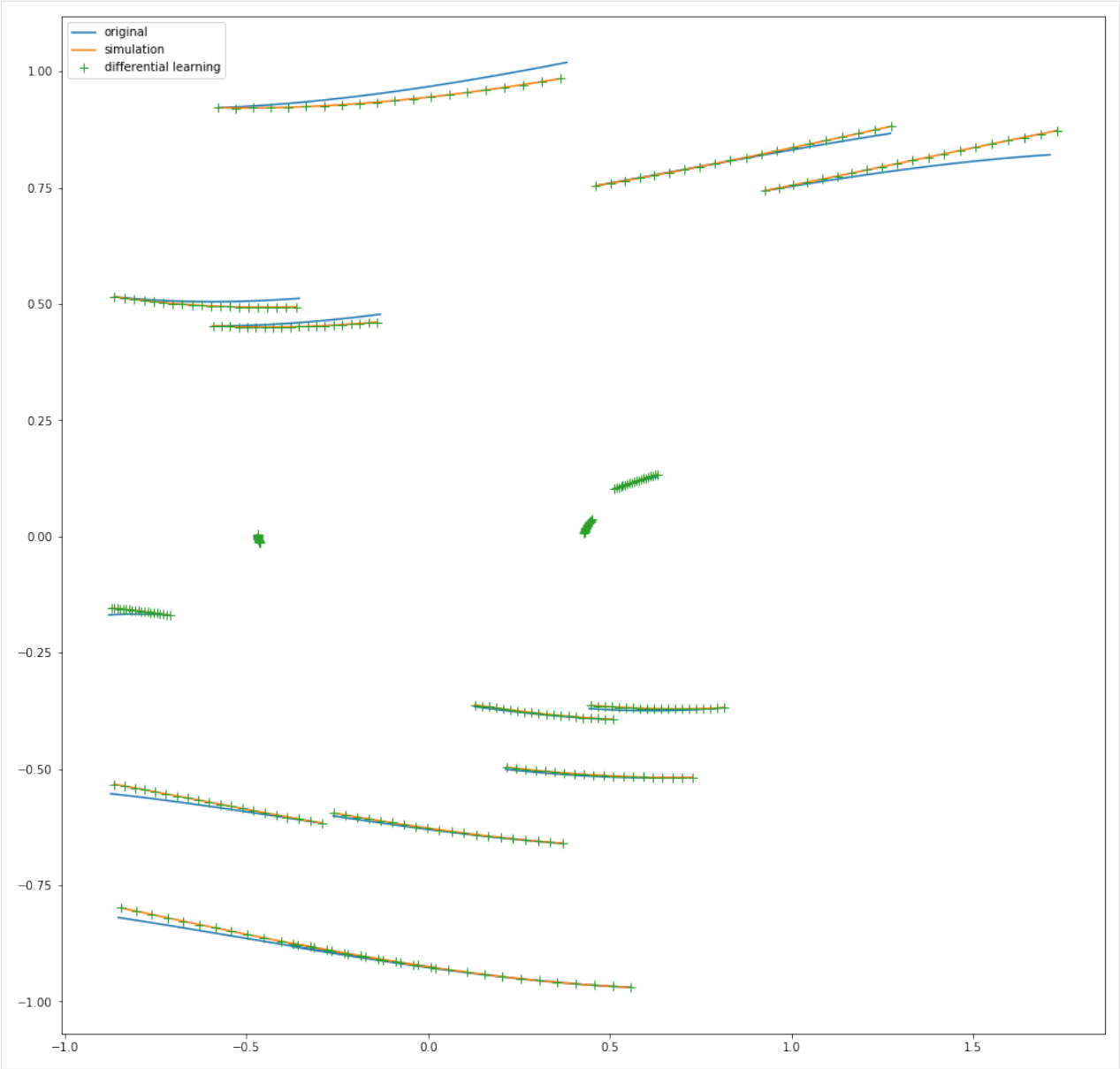
We now simulate again our behavior using the new weights to see where we end up to check if the prediction made after our differential learning is correct.

[11]:
```
new_weights = D.array([it + dweights['d' + it.symbol_set[0]] for it in weights])
```

[12]:
```
pyaudi_integration2 = de.OdeSystem(rhs, y0=y0, dense_output=True, t=(pyaudi_
↪integration.t[0], pyaudi_integration.t[-1]), dt=0.1, rtol=1e-12, atol=1e-12,␣
↪constants=dict(weights=new_weights))

pyaudi_integration2.set_method("RK45")
pyaudi_integration2.integrate(eta=True)
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

[13]:
```
plt.figure(figsize=(16,16))
x2, v2 = pyaudi_integration2.sol(D.linspace(0, 1, 20)).T
for idx, (_x, _v) in enumerate(zip(D.to_float(x).T, D.to_float(v).T)):
    if idx == 0:
        plt.plot(_x,_v,'C0',label='original')
    else:
        plt.plot(_x,_v,'C0')
for idx, (_x, _v) in enumerate(zip(D.to_float(x2).T, D.to_float(v2).T)):
    if idx == 0:
        plt.plot(_x,_v,'C1',label='simulation')
    else:
        plt.plot(_x,_v,'C1')
for idx, (_x, _v) in enumerate(zip(D.array([it.evaluate(dweights) for it in x]).T,D.
↪array([it.evaluate(dweights) for it in v]).T)):
    if idx == 0:
        plt.plot(_x,_v,'C2+', markersize=8.,label='differential learning')
    else:
        plt.plot(_x,_v,'C2+', markersize=8.)
# plt.plot(x[0].constant_cf, v[0].constant_cf, 'ro')
plt.legend(loc=2)
plt.show()
```



Since we have integrated multiple trajectories, printing all the final states and comparing it to the evaluated polynomial is visually difficult to parse. Instead, we look at the maximum and mean absolute differences between the numerical integration and the map evaluated with the new weights.

[14]:
```
print("Maximum Absolute Difference xf:\t{}".format(D.max(D.abs(D.to_float(x2[-1]) - D.
↪to_float(x[-1].evaluate(dweights))))))
print("Mean Absolute Difference xf:   \t{}".format(D.mean(D.abs(D.to_float(x2[-1]) -␣
↪D.to_float(x[-1].evaluate(dweights))))))
print()
print("Maximum Absolute Difference vf:\t{}".format(D.max(D.abs(D.to_float(v2[-1]) - D.
↪to_float(v[-1].evaluate(dweights))))))
print("Mean Absolute Difference vf:   \t{}".format(D.mean(D.abs(D.to_float(v2[-1]) -␣
↪D.to_float(v[-1].evaluate(dweights))))))
```

```
Maximum Absolute Difference xf: 3.121884417645049e-10
Mean Absolute Difference xf:    4.664606374771285e-11
```

(continues on next page)

(continued from previous page)

```
Maximum Absolute Difference vf: 9.84426262728988e-10
Mean Absolute Difference vf:    1.7914556643683355e-10
```

6.5.3 Differential Intelligence - Quadruple Precision Integration

(original by Dario Izzo - extended by Ekin Ozturk)

In this notebook we show the use of `desolver` and quadruple precision arithmetic for numerical integration following the notebook example here differential intelligence.

(`gdual_real128` is not available on every platform, consult the PyAudi documentation here)

Importing Stuff

[1]:
```
%matplotlib inline
from matplotlib import pyplot as plt

import os
import numpy as np

os.environ['DES_BACKEND'] = 'numpy'
import desolver as de
import desolver.backend as D

D.set_float_fmt('gdual_real128')
```

```
PyAudi backend is available.
Using numpy backend
```

Controller representation and "simulator"

Take as an example the task of learning a robotic controller. In neuro evolution (or Evolutionary Robotics), the controller is typically represented by a neural network, but for the purpose of explaining this new learning concept we will use a polinomial representation for the controller. Later, changing the controller into a NN with weights as parameters will not change the essence of what is done here.

[2]:
```
# Definition of the controller in terms of some weights parameters
def u(state, weights):
    x,v = state
    a,b,c,e,f,g = weights
    return a + b*x + c*v + e*x*v + f*x**2 + g*v**2
```

[3]:
```
# Definition of the equation of motion (our physics simulator propagating the system␣
↪to its next state)
def eom(state, weights):
    x,v = state
    dx = v
    dv = u(state, weights)
    return (dx, dv)
```

Numerical Integration - Runge-Kutta 8(7) Dormand-Prince Method

In Evolutionary Robotics, Euler propagators are commonly used, but we would like to use a higher order integration scheme that is adaptive in order to minimise computation, and increase the accuracy and precision of the results.

[4]:
```
weights = D.array([D.gdual_real128(0.05*(np.random.uniform()-0.5), _, 4) for _ in
↪"abcefg"])
x = [D.gdual_real128(2*(np.random.uniform()-0.5))]
v = [D.gdual_real128(2*(np.random.uniform()-0.5))]
y0 = D.array(x + v, dtype=D.gdual_real128)
```

[5]:
```
def rhs(t, state, weights, **kwargs):
    return D.array(eom(state, weights))
```

We integrate the system using the Runge-Kutta 8(7) scheme[1] as the numerical integration system with a dense output computed using a piecewise C1 Hermite interpolating spline.

This particular interpolator is used as it satisfies not only the state boundary conditions, but also the gradients and is well suited for approximating the solution continuously up to first order in time.

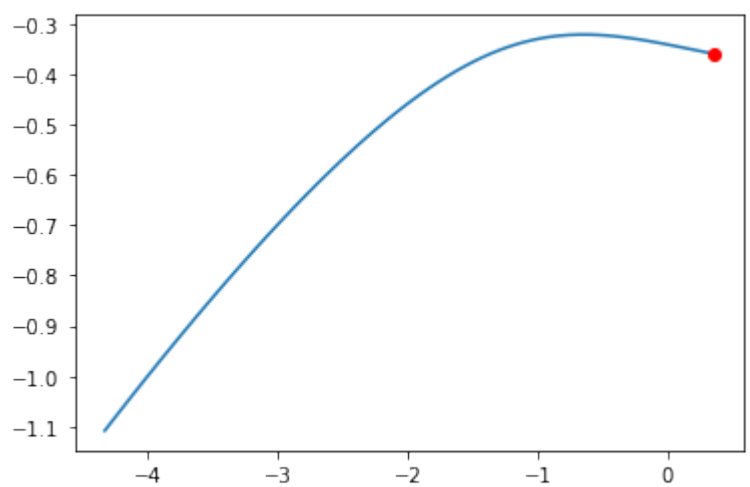
Note that the entire integration is done using gduals and thus the solution and the interpolating spline stored in the OdeSystem instance, `pyaudi_integration`, contains all the information about how the state reached by our robot changes when we change the control weights.

[1] Prince, P.J., and J.R. Dormand. 'High Order Embedded Runge-Kutta Formulae'. Journal of Computational and Applied Mathematics 7, no. 1 (March 1981): 67–75. https://doi.org/10.1016/0771-050X(81)90010-3.

[6]:
```
pyaudi_integration = de.OdeSystem(rhs, y0=y0, dense_output=False, t=(0, 10.), dt=0.1,␣
↪rtol=1e-16, atol=1e-16, constants=dict(weights=weights))

pyaudi_integration.set_method("RK87")
pyaudi_integration.integrate(eta=True)
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

[7]:
```
x,v = pyaudi_integration.y.T
plt.plot([it for it in x.astype(np.float64)],[it for it in v.astype(np.float64)])
plt.plot(x.astype(np.float64)[0], v.astype(np.float64)[0], 'ro')
plt.show()
```



[8]:
```
xf, vf = x[-1], v[-1]
```

[9]:
```
print("initial xf: {}".format(xf.constant_cf))
print("initial vf: {}".format(vf.constant_cf))
```

```
initial xf: -4.326435190339891798190747099407153 37e+00
initial vf: -1.108100729403454502336138627271914 35e+00
```

Studying the effects of the weights on the behavior

We have represented all the robot behavior (x, v) as a polynomial function of the weights. So we now know what happens to the behaviour if we change the weights!! Lets see … we only consider the final state, but the same can be done for all states before.

[10]:
```
dweights = dict({'da': -0.0002, 'db': 0.0003, 'dc': -0.002, 'de': 0.003, 'df': 0.002,
↪'dg': -0.001})
#Lets predict the new final position of our 'robot' if we change his controller as␣
↪defined above
print("new xf: {}".format(xf.evaluate(dweights)))
print("new vf: {}".format(vf.evaluate(dweights)))
```

```
new xf: -4.087901053485909793431135719608533 45e+00
new vf: -9.760254040801188882521957343253483 74e-01
```

Check that we learned the correct map

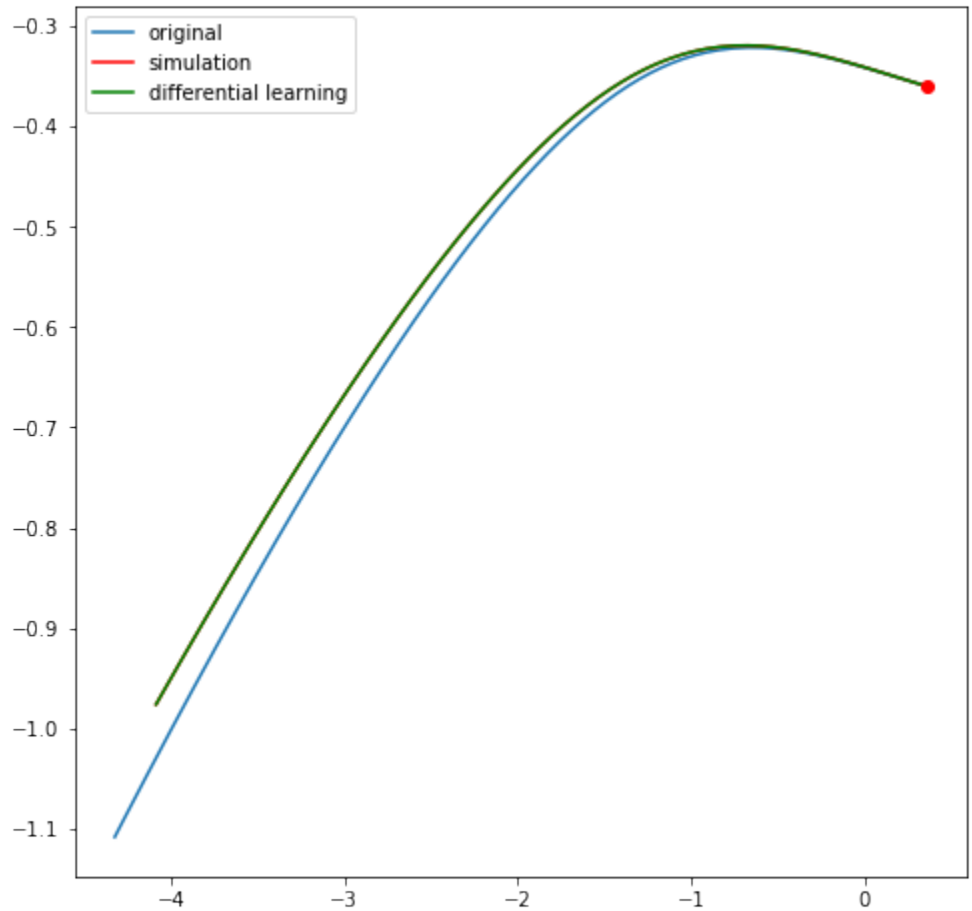
We now simulate again our behavior using the new weights to see where we end up to check if the prediction made after our differential learning is correct.

[11]:
```
new_weights = np.array([it + dweights['d' + it.symbol_set[0]] for it in weights])
```

[12]:
```
pyaudi_integration2 = de.OdeSystem(rhs, y0=y0, dense_output=True, t=(0, 10.), dt=0.01,
↪ rtol=1e-16, atol=1e-16, constants=dict(weights=new_weights))

pyaudi_integration2.set_method("RK87")
pyaudi_integration2.integrate(eta=True)
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

[13]:
```
plt.figure(figsize=(8,8))
x2, v2 = pyaudi_integration2.y.T
plt.plot([it for it in x.astype(np.float64)],[it for it in v.astype(np.float64)],
↪label='original')
plt.plot([it for it in x2.astype(np.float64)],[it for it in v2.astype(np.float64)],'r
↪',label='simulation')
plt.plot([float(str(it.evaluate(dweights))) for it in x],[float(str(it.
↪evaluate(dweights))) for it in v],'g',label='differential learning')
plt.plot(x.astype(np.float64)[0], v.astype(np.float64)[0], 'ro')
plt.legend(loc=2)
plt.show()
```



[14]:
```
print("Differential learning xf: \t{}".format(x[-1].evaluate(dweights)))
print("Real xf: \t\t\t{}".format(x2[-1].constant_cf))
print("Mean Absolute Difference xf:\t{}".format(x2[-1].constant_cf - x[-1].
↪evaluate(dweights)))
print()
print("Differential learning vf: \t{}".format(v[-1].evaluate(dweights)))
print("Real vf: \t\t\t{}".format(v2[-1].constant_cf))
print("Mean Absolute Difference vf:\t{}".format(v2[-1].constant_cf - v[-1].
↪evaluate(dweights)))
```

```
Differential learning xf:       -4.0879010534859097934311357196085334 5e+00
Real xf:                        -4.0879004972189819882670909182379419 2e+00
Mean Absolute Difference xf:    5.5626692780516404480137059153115500 8e-07

Differential learning vf:       -9.7602540408011888825219573432534837 4e-01
Real vf:                        -9.7602446444469043563336266054130414 5e-01
Mean Absolute Difference vf:    9.3963542845261883307378404422880431 7e-07
```

[15]:
```
pyaudi_integration3 = de.OdeSystem(rhs, y0=y0, dense_output=True, t=(0, 10.), dt=0.01,
↪ rtol=1e-16, atol=1e-16, constants=dict(weights=new_weights))

pyaudi_integration3.set_method("RK45")
pyaudi_integration3.integrate(eta=True)

x3, v3 = pyaudi_integration3.y.T
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

[16]:
```
print("Difference between RK45 and RK87 xf: \t{}".format((x2[-1] - x3[-1]).constant_
↪cf))
print("Difference between RK45 and RK87 vf: \t{}".format((v2[-1] - v3[-1]).constant_
↪cf))
print()
print("Mean Absolute Difference xf[RK87]:\t{}".format(x2[-1].constant_cf - x[-1].
↪evaluate(dweights)))
print("Mean Absolute Difference xf[RK45]:\t{}".format(x3[-1].constant_cf - x[-1].
↪evaluate(dweights)))
print()
print("Mean Absolute Difference vf[RK87]:\t{}".format(v2[-1].constant_cf - v[-1].
↪evaluate(dweights)))
print("Mean Absolute Difference vf[RK45]:\t{}".format(v3[-1].constant_cf - v[-1].
↪evaluate(dweights)))
```

```
Difference between RK45 and RK87 xf:    5.735707012363626440526568113604999 30e-15
Difference between RK45 and RK87 vf:    1.981825201633754839571279031017988 99e-15

Mean Absolute Difference xf[RK87]:      5.562669278051640448013705915311550 08e-07
Mean Absolute Difference xf[RK45]:      5.562669220694570324377441510045868 94e-07

Mean Absolute Difference vf[RK87]:      9.396354284526188330737840442288043 17e-07
Mean Absolute Difference vf[RK45]:      9.396354264707936314400292046575252 86e-07
```

[ ]:

[ ]:

6.5.4 High Order Taylor Maps I

(original by Dario Izzo - extended by Ekin Ozturk)

Building upon the notebook here, we show the use of `desolver` for numerically integrating the system of differential equations $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$:

$$\begin{aligned}
\dot{r} &= v_r \\
\dot{v}_r &= -\frac{1}{r^2} + r v_\theta^2 \\
\dot{\theta} &= v_\theta \\
\dot{v}_\theta &= -2\frac{v_\theta v_r}{r}
\end{aligned}$$

which describe, in non dimensional units, the Keplerian motion of a mass point object around some primary body. We show how we can build a high order Taylor map (HOTM, indicated with $\mathcal{M}$) representing the final state of the system at the time T as a function of the initial conditions.

In other words, we build a polinomial representation of the relation $\mathbf{y}(T) = \mathbf{f}(\mathbf{y}(0), T)$. Writing the initial conditions as $\mathbf{y}(0) = \overline{\mathbf{y}}(0) + \mathbf{dy}$, our HOTM will be written as:

$$\mathbf{y}(T) = \mathcal{M}(\mathbf{dy})$$

and will be valid in a neighbourhood of $\overline{\mathbf{y}}(0)$.

Importing Stuff

[1]:
```
%matplotlib inline
from matplotlib import pyplot as plt

import os
import numpy as np

os.environ['DES_BACKEND'] = 'numpy'
import desolver as de
import desolver.backend as D
from desolver.backend import gdual_double as gdual
```

```
PyAudi backend is available.
Using numpy backend
```

[2]:
```
T = 1e-3

@de.rhs_prettifier(equ_repr="[vr, -1/r**2 + r*vt**2, vt, -2*vt*vr/r]", md_repr=r"""$$
\begin{array}{l}
\dot r = v_r \\
\dot v_r = - \frac 1{r^2} + r v_\theta^2\\
\dot \theta = v_\theta \\
\dot v_\theta = -2 \frac{v_\theta v_r}{r}
\end{array}
$$""")
def eom_kep_polar(t,y,**kwargs):
    return D.array([y[1], - 1 / y[0] / y[0] + y[0] * y[3]*y[3], y[3], -2*y[3]*y[1]/
↪y[0] - T])

eom_kep_polar
```

[2]:

$$\begin{array}{l}
\dot r = v_r \\
\dot v_r = - \frac 1{r^2} + r v_\theta^2\\
\dot \theta = v_\theta \\
\dot v_\theta = -2 \frac{v_\theta v_r}{r}
\end{array}$$

[3]:
```
# The initial conditions
ic = [1.,0.1,0.,1.]
```

We perform the numerical integration using floats (the standard way)

[4]:
```
D.set_float_fmt('float64')
float_integration = de.OdeSystem(eom_kep_polar, y0=ic, dense_output=False, t=(0, 5.),␣
↪dt=0.01, rtol=1e-12, atol=1e-12, constants=dict())

float_integration.set_method("RK45")
float_integration.integrate(eta=True)
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

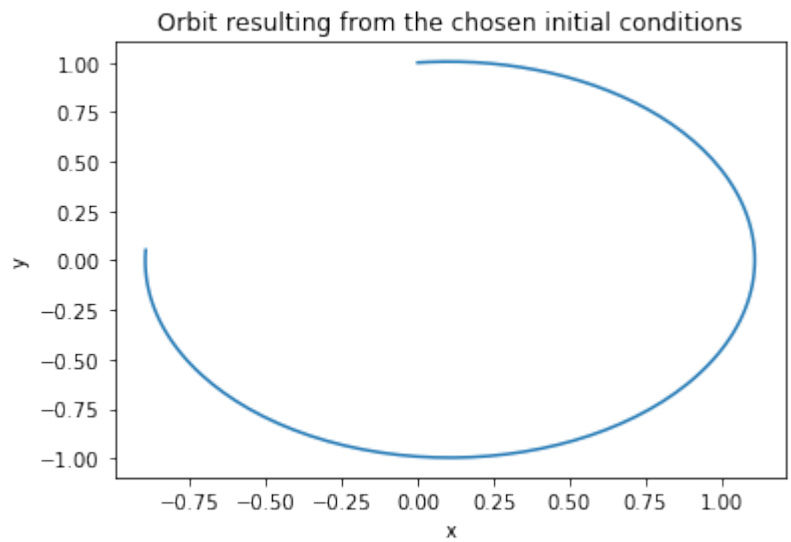
[5]:
```
# Here we transform from polar to cartesian coordinates
# to then plot
```

(continues on next page)

(continued from previous page)

```
y = float_integration.y
cx = [it[0]*np.sin(it[2]) for it in y.astype(np.float64)]
cy = [it[0]*np.cos(it[2]) for it in y.astype(np.float64)]
plt.plot(cx,cy)
plt.title("Orbit resulting from the chosen initial conditions")
plt.xlabel("x")
plt.ylabel("y")
```

[5]:
```
Text(0, 0.5, 'y')
```



We perform the numerical integration using gduals (to get a HOTM)

[6]:
```
# Order of the Taylor Map. If we have 4 variables the number of terms in the Taylor␣
↪expansion in 329  at order 7
order = 5
# We now define the initial conditions as gdual (not float)
ic_g = [gdual(ic[0], "r", order), gdual(ic[1], "vr", order), gdual(ic[2], "t", order),
↪ gdual(ic[3], "vt", order)]
```

[7]:
```
import time
start_time = time.time()
D.set_float_fmt('gdual_double')
gdual_integration = de.OdeSystem(eom_kep_polar, y0=ic_g, dense_output=False, t=(0, 5.
↪), dt=0.01, rtol=1e-12, atol=1e-12, constants=dict())

gdual_integration.set_method("RK45")
gdual_integration.integrate(eta=True)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

```
--- 3.7148001194000244 seconds ---
```

[8]:
```
# We extract the last point
yf = gdual_integration.y[-1]
```

(continues on next page)

(continued from previous page)

```
# And unpack it into some convinient names
rf,vrf,tf,vtf = yf
# We compute the final cartesian components
xf = rf * D.sin(tf)
yf = rf * D.cos(tf)
# Note that you can get the latex representation of the gdual
print(xf._repr_latex_())
print("xf (latex):")
xf
```

```
\[ 9.62324{dt}^{3}{dvt}^{2}-287.941{dt}^{2}{dvt}^{3}-339.246{dr}{dt}^{2}{dvr}^{2}-
↪1163.38{dr}^{2}{dt}^{2}{dvr}-55841{dr}^{3}{dvr}-14155.1{dvt}^{4}+173.989{dvt}^{2}
↪+45397.4{dr}^{4}{dt}-2.51218e+06{dr}^{2}{dvt}^{3}-36.4141{dt}^{2}{dvr}{dvt}+2655.96
↪{dr}^{3}+8351.27{dr}{dt}{dvt}^{2}+134833{dr}^{3}{dt}{dvt}-4.31954e+06{dr}^{3}{dvt}^
↪{2}-2.94774e+06{dr}^{2}{dvr}{dvt}^{2}-13.1439{dt}{dvr}^{2}+101890{dr}^{2}{dt}{dvr}
↪{dvt}+22840.3{dr}{dt}{dvr}^{2}{dvt}+2550.6{dr}{dvr}{dvt}+5780.38{dr}^{2}{dvt}+346.
↪756{dvr}^{2}{dvt}+1330.3{dt}{dvr}^{3}{dvt}-20.4879{dt}{dvt}+0.631304{dt}^{3}{dvr}-
↪1710.37{dr}^{2}{dvr}^{2}-0.110791{dt}^{4}{dvt}-337.535{dr}{dt}^{2}{dvt}+762.507{dt}
↪{dvr}{dvt}^{2}-64.2798{dr}{dt}{dvr}-25920.5{dvr}^{3}{dvt}^{2}+138.386{dvr}^{4}+0.
↪633023{dvr}+83.5605{dt}{dvr}^{2}{dvt}+2892.65{dr}{dt}{dvr}{dvt}+72.8282{dvr}{dvt}-0.
↪316512{dt}^{2}{dvr}+5.3273{dt}^{3}{dvr}{dvt}+46793.3{dr}{dt}{dvt}^{3}+141.812{dr}
↪{dvr}+10.7133{dr}{dt}^{3}{dvr}-3.60855e+06{dr}^{3}{dvr}{dvt}-24.6985{dt}^{2}{dvr}^
↪{3}-545726{dr}{dvr}^{2}{dvt}^{2}+1260.23{dr}{dvr}^{3}+2821.14{dr}^{2}{dt}{dvr}-
↪95997.3{dvr}^{2}{dvt}^{3}+6018.59{dt}{dvt}^{4}-39.0601{dr}{dt}-0.219359{dr}{dt}^{4}-
↪86.9944{dt}^{2}{dvt}^{2}+\ldots+\mathcal{O}\left(6\right) \]
xf (latex):
```

[8]:

$$9.62324dt^3dvt^2-287.941dt^2dvt^3-339.246drdt^2dvr^2-1163.38dr^2dt^2dvr-55841dr^3dvr-14155.1dvt^4+173.989dvt^2+45397.4dr^4d$$

[9]:
```
# We can extract the value of the polinomial when $\mathbf {dy} = 0$
print("Final x from the gdual integration", xf.constant_cf)
print("Final y from the gdual integration", yf.constant_cf)
# And check its indeed the result of the 'reference' trajectory (the lineariation␣
↪point)
print("\nFinal x from the float integration", cx[-1])
print("Final y from the float integration", cy[-1])
```

```
Final x from the gdual integration -0.8953272292032805
Final y from the gdual integration 0.052709201662514145

Final x from the float integration -0.8953272292032807
Final y from the float integration 0.052709201662513354
```

We visualize the HOTM

[10]:
```
# Let us now visualize the Taylor map by creating a grid of perturbations on the␣
↪initial conditions and
# evaluating the map for those values
Npoints = 10 # 10000 points
epsilon = 1e-3
grid = np.arange(-epsilon,epsilon,2*epsilon/Npoints)
nxf = [0] * len(grid)**4
nyf = [0] * len(grid)**4
i=0
```

(continues on next page)

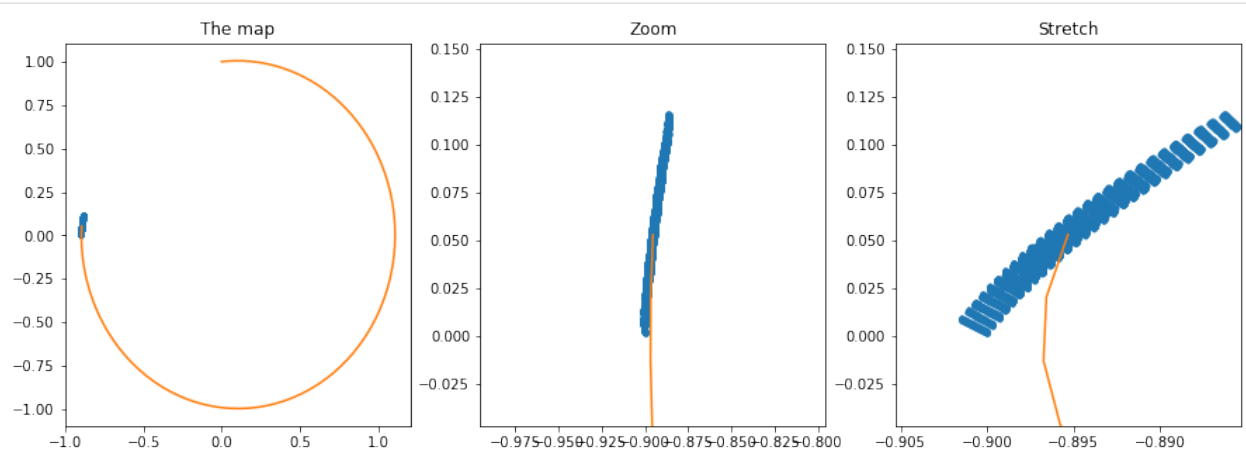
(continued from previous page)

```
import time
start_time = time.time()
for dr in grid:
    for dt in grid:
        for dvr in grid:
            for dvt in grid:
                nxf[i] = xf.evaluate({"dr":dr, "dt":dt, "dvr":dvr,"dvt":dvt})
                nyf[i] = yf.evaluate({"dr":dr, "dt":dt, "dvr":dvr,"dvt":dvt})
                i = i+1
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 0.9997782707214355 seconds ---
```

[11]:
```
f, axarr = plt.subplots(1,3,figsize=(15,5))
# Normal plot of the final map
axarr[0].plot(nxf,nyf,'.')
axarr[0].plot(cx,cy)
axarr[0].set_title("The map")

# Zoomed plot of the final map (equal axis)
axarr[1].plot(nxf,nyf,'.')
axarr[1].plot(cx,cy)
axarr[1].set_xlim([cx[-1] - 0.1, cx[-1] + 0.1])
axarr[1].set_ylim([cy[-1] - 0.1, cy[-1] + 0.1])
axarr[1].set_title("Zoom")

# Zoomed plot of the final map (unequal axis)
axarr[2].plot(nxf,nyf,'.')
axarr[2].plot(cx,cy)
axarr[2].set_xlim([cx[-1] - 0.01, cx[-1] + 0.01])
axarr[2].set_ylim([cy[-1] - 0.1, cy[-1] + 0.1])
axarr[2].set_title("Stretch")
#axarr[1].set_xlim([cx[-1] - 0.1, cx[-1] + 0.1])
#axarr[1].set_ylim([cy[-1] - 0.1, cy[-1] + 0.1])
```

[11]:
```
Text(0.5, 1.0, 'Stretch')
```



How much faster is now to evaluate the Map rather than perform a new numerical integration?

[12]:
```
# First we profile the method evaluate (note that you need to call the method 4 times␣
↪to get the full state)
```

[13]:
```
%timeit xf.evaluate({"dr":epsilon, "dt":epsilon, "dvr":epsilon,"dvt":epsilon})
```

```
50.3 µs ± 4.94 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

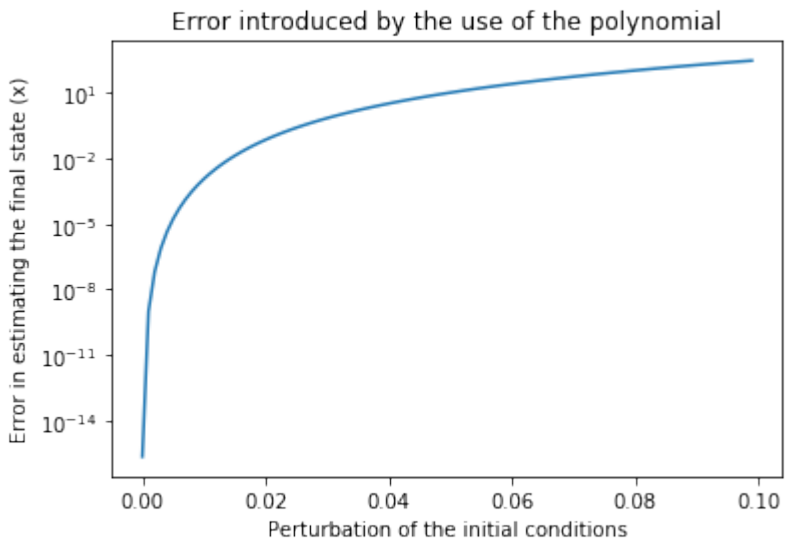
[14]:
```
# Then we profile the Runge-Kutta 4 integrator
```

[15]:
```
%%timeit
D.set_float_fmt('float64')
float_integration = de.OdeSystem(eom_kep_polar, y0=[it + epsilon for it in ic], dense_
↪output=False, t=(0, 5.), dt=0.01, rtol=1e-12, atol=1e-12, constants=dict())

float_integration.set_method("RK45")
float_integration.integrate(eta=False)
```

```
75.1 ms ± 7.15 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[16]:
```
# It seems the speedup is 2-3 orders of magnitude, but did we loose precision?
# We plot the error in the final result as computed by the HOTM and by the Runge-Kutta
# as a function of the distance from the original initial conditions
out = []
pert = np.arange(0,0.1,1e-3)
for epsilon in pert:
    res_map_xf = xf.evaluate({"dr":epsilon, "dt":epsilon, "dvr":epsilon,"dvt":epsilon}
↪)
    res_int = de.OdeSystem(eom_kep_polar, y0=[it + epsilon for it in ic], dense_
↪output=False, t=(0, 5.), dt=0.01, rtol=1e-12, atol=1e-12, constants=dict())
    res_int.set_method("RK45")
    res_int.integrate()
    res_int_x = [it.y[0]*np.sin(it.y[2]) for it in res_int]
    res_int_xf = res_int_x[-1]
    out.append(np.abs(res_map_xf - res_int_xf))
plt.semilogy(pert,out)
plt.title("Error introduced by the use of the polynomial")
plt.xlabel("Perturbation of the initial conditions")
plt.ylabel("Error in estimating the final state (x)")
```

[16]:
```
Text(0, 0.5, 'Error in estimating the final state (x)')
```



6.5.5 High Order Taylor Maps II

(original by Dario Izzo - extended by Ekin Ozturk)

Building upon the notebook here, we show the use of `desolver` for numerically integrating the system of differential equations $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$:

$$\begin{aligned}
\dot{r} &= v_r \\
\dot{v}_r &= -\frac{1}{r^2} + r v_\theta^2 \\
\dot{\theta} &= v_\theta \\
\dot{v}_\theta &= -2\frac{v_\theta v_r}{r} + T
\end{aligned}$$

which describe, in non dimensional units, the motion of a mass point object around some primary body perturbed by a fixed thrust T acting in the direction perpendicular to the radius vector. We show how we can build a high order Taylor map (HOTM, indicated with $\mathcal{M}$) representing the final state of the system at the time T as a function of the initial conditions.

In other words, we build a polinomial representation of the relation $\mathbf{y}(T) = \mathbf{f}(\mathbf{y}(0), T)$. Writing the initial conditions as $\mathbf{y}(0) = \overline{\mathbf{y}}(0) + \mathbf{dy}$, our HOTM will be written as:

$$\mathbf{y}(T) = \mathcal{M}(\mathbf{dy})$$

and will be valid in a neighbourhood of $\overline{\mathbf{y}}(0)$.

Importing Stuff

[1]:
```
%matplotlib inline
from matplotlib import pyplot as plt

import os
import numpy as np

os.environ['DES_BACKEND'] = 'numpy'
import desolver as de
import desolver.backend as D
from desolver.backend import gdual_double as gdual
```

```
PyAudi backend is available.
Using numpy backend
```

[2]:
```
T = 1e-3

@de.rhs_prettifier(equ_repr="[vr, -1/r**2 + r*vt**2, vt, -2*vt*vr/r]", md_repr=r"""$$
\begin{array}{l}
\dot r = v_r \\
\dot v_r = - \frac 1{r^2} + r v_\theta^2\\
\dot \theta = v_\theta \\
\dot v_\theta = -2 \frac{v_\theta v_r}{r}
\end{array}
$$""")
def eom_kep_polar(t,y,**kwargs):
    return D.array([y[1], - 1 / y[0] / y[0] + y[0] * y[3]*y[3], y[3], -2*y[3]*y[1]/
↪y[0] - T])

eom_kep_polar
```

[2]:

$$\begin{array}{l}
\dot r = v_r \\
\dot v_r = - \frac 1{r^2} + r v_\theta^2\\
\dot \theta = v_\theta \\
\dot v_\theta = -2 \frac{v_\theta v_r}{r}
\end{array}$$

[3]:
```
# The initial conditions
ic = [1.,0.1,0.,-1.]
```

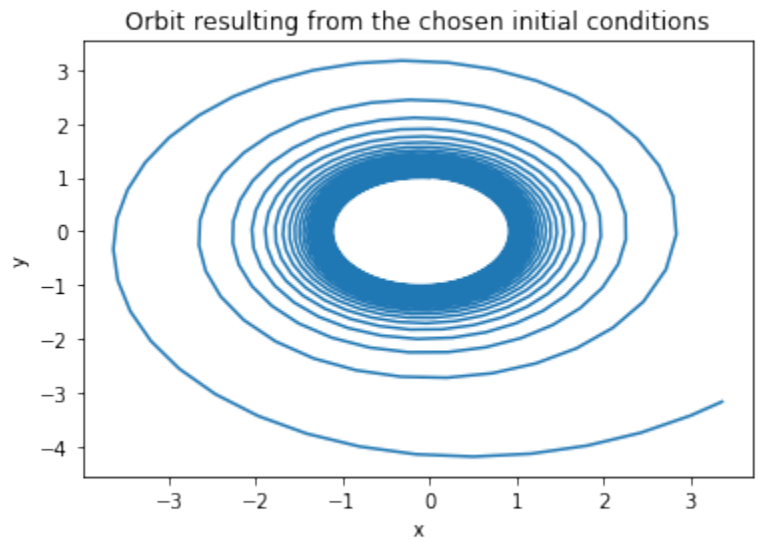
We perform the numerical integration using floats (the standard way)

[4]:
```
D.set_float_fmt('float64')
float_integration = de.OdeSystem(eom_kep_polar, y0=ic, dense_output=False, t=(0, 300.
↪), dt=0.01, rtol=1e-10, atol=1e-10, constants=dict())

float_integration.set_method("RK45")
float_integration.integrate(eta=True)
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

[5]:
```
# Here we transform from polar to cartesian coordinates
# to then plot
y = float_integration.y
cx = [it[0]*np.sin(it[2]) for it in y.astype(np.float64)]
cy = [it[0]*np.cos(it[2]) for it in y.astype(np.float64)]
plt.plot(cx,cy)
plt.title("Orbit resulting from the chosen initial conditions")
plt.xlabel("x")
plt.ylabel("y")
```

[5]:
```
Text(0, 0.5, 'y')
```



We perform the numerical integration using gduals (to get a HOTM)

[6]:
```
# Order of the Taylor Map. If we have 4 variables the number of terms in the Taylor␣
↪expansion in 329  at order 7
order = 6
# We now define the initial conditions as gdual (not float)
ic_g = [gdual(ic[0], "r", order), gdual(ic[1], "vr", order), gdual(ic[2], "t", order),
↪ gdual(ic[3], "vt", order)]
```

[7]:
```
import time
start_time = time.time()
D.set_float_fmt('gdual_double')
gdual_integration = de.OdeSystem(eom_kep_polar, y0=ic_g, dense_output=False, t=(0,␣
↪300.), dt=0.01, rtol=1e-10, atol=1e-10, constants=dict())

gdual_integration.set_method("RK45")
gdual_integration.integrate(eta=True)
print("--- %s seconds ---" % (time.time() - start_time))
```

```
HBox(children=(FloatProgress(value=0.0, max=9000000000.0), HTML(value='')))
```

```
--- 38.72461462020874 seconds ---
```

[8]:
```
# We extract the last point
yf = gdual_integration.y[-1]
# And unpack it into some convinient names
rf,vrf,tf,vtf = yf
# We compute the final cartesian components
xf = rf * D.sin(tf)
yf = rf * D.cos(tf)
# Note that you can get the latex representation of the gdual
print(xf._repr_latex_())
print("xf (latex):")
xf
```

```
\[ 5.27792e+09{dvr}^{2}{dvt}^{2}+2.38683e+07{dr}{dt}{dvr}^{2}+812955{dr}{dt}^{3}{dvt}
↪+0.528723{dt}^{3}-3.96318e+08{dvr}^{3}{dvt}-4.87979e+09{dr}^{2}{dt}{dvt}-1.17038e+16
↪{dr}{dvr}{dvt}^{4}-0.00467575{dt}^{6}+6.75558e+13{dr}{dvr}{dvt}^{3}+1.41974e+12{dr}^
↪{3}{dt}^{2}{dvt}+1.05585e+10{dr}{dt}^{2}{dvr}^{2}{dvt}-4.74308e+17{dr}^{4}{dvt}^{2}-
↪1.9878e+15{dvt}^{6}+1.7157e+13{dr}^{2}{dvr}^{2}{dvt}-2.09573e+12{dr}^{2}{dt}{dvt}^
↪{2}+1.43888e+15{dr}^{4}{dvt}+2.20856e+09{dr}{dvt}^{2}+2.75642e+06{dr}^{2}{dt}^{2}-1.
↪97312e+11{dr}{dt}{dvr}{dvt}^{2}-5.40989e+08{dr}^{3}{dt}^{3}+1.03991e+08{dvr}{dvt}^
↪{2}-1.91427e+07{dt}^{3}{dvr}{dvt}^{2}+3.36654-1.92511e+13{dr}^{2}{dvr}^{4}+689698
↪{dt}^{2}{dvt}^{2}-4.07038e+08{dr}{dt}^{3}{dvt}^{2}-3.93014e+09{dvr}^{6}-6200.56{dr}
↪{dt}-8.73108e+10{dt}{dvt}^{4}+1.43057e+12{dvr}^{2}{dvt}^{3}+4.59063e+08{dr}^{2}{dt}
↪{dvr}-230.032{dr}{dt}^{4}+7.92742e+08{dr}{dvr}^{3}-8.57426e+12{dr}{dvr}^{2}{dvt}^{2}
↪-2.63896e+09{dt}^{2}{dvr}^{2}{dvt}^{2}-93896{dt}^{2}{dvr}^{3}+1.16891e+15{dvr}{dvt}^
↪{5}+2.01684e+11{dr}^{2}{dt}^{2}{dvr}{dvt}-1.34155e+11{dr}^{3}{dt}^{2}{dvr}+2.
↪70709e+14{dr}^{3}{dvr}{dvt}-7.0871e+11{dr}^{2}{dt}{dvr}^{3}-518252{dr}{dvr}-1.
↪19378e+17{dr}^{2}{dvt}^{4}+6.98014e+11{dr}{dt}{dvt}^{3}-76467.7{dr}{dt}^{3}{dvr}+7.
↪09031e+11{dr}{dt}{dvr}^{3}{dvt}-4.46087e+11{dr}{dvr}^{5}+38235.3{dt}^{3}{dvr}{dvt}
↪+130.196{dt}^{2}{dvr}-2.42424{dt}^{5}{dvr}+\ldots+\mathcal{O}\left(7\right) \]
xf (latex):
```

[8]:

$$5.27792e+09dvr^2dvt^2+2.38683e+07drdtdvr^2+812955drdt^3dvt+0.528723dt^3-3.96318e+08dvr^3dvt-4.87979e+09dr^2dtdvt-1.$$

[9]:
```
# We can extract the value of the polinomial when $\mathbf {dy} = 0$
print("Final x from the gdual integration", xf.constant_cf)
print("Final y from the gdual integration", yf.constant_cf)
# And check its indeed the result of the 'reference' trajectory (the lineariation␣
↪point)
print("\nFinal x from the float integration", cx[-1])
print("Final y from the float integration", cy[-1])
```

```
Final x from the gdual integration 3.366536670599516
Final y from the gdual integration -3.172339703511845

Final x from the float integration 3.366536670599606
Final y from the float integration -3.1723397035117493
```

We visualize the HOTM

[10]:
```
# Let us now visualize the Taylor map by creating a grid of perturbations on the␣
↪initial conditions and
# evaluating the map for those values
Npoints = 20 # 10000 points
epsilon = 1e-3
grid = np.arange(-epsilon,epsilon,2*epsilon/Npoints)
nxf = [0] * len(grid)**4
nyf = [0] * len(grid)**4
i=0
import time
start_time = time.time()
for dr in grid:
    for dt in grid:
        for dvr in grid:
            for dvt in grid:
                nxf[i] = xf.evaluate({"dr":dr, "dt":dt, "dvr":dvr,"dvt":dvt})
                nyf[i] = yf.evaluate({"dr":dr, "dt":dt, "dvr":dvr,"dvt":dvt})
```

(continues on next page)

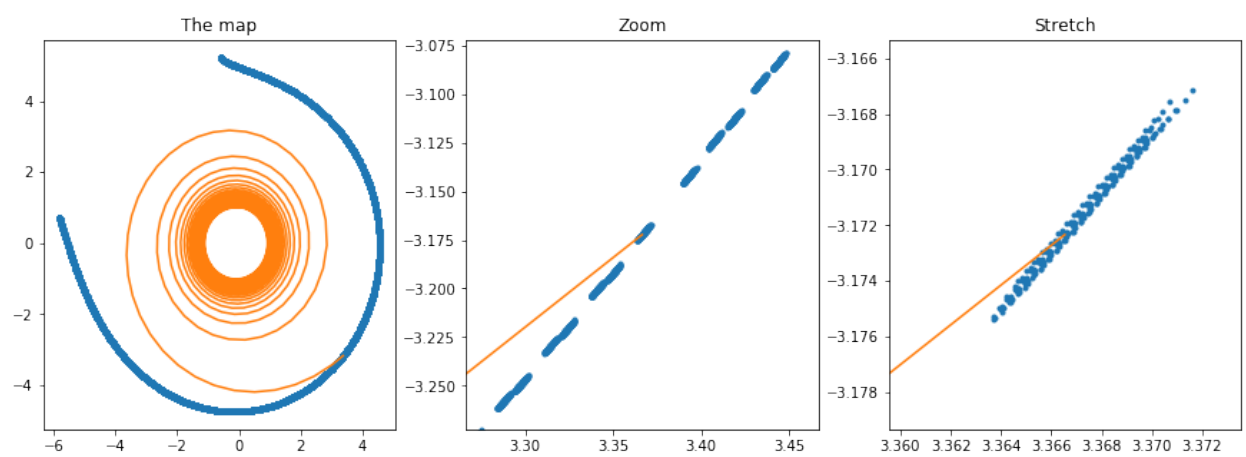
(continued from previous page)

```
                i = i+1
print("--- %s seconds ---" % (time.time() - start_time))
```

```
--- 23.139215230941772 seconds ---
```

[11]:
```
f, axarr = plt.subplots(1,3,figsize=(15,5))
# Normal plot of the final map
axarr[0].plot(nxf,nyf,'.')
axarr[0].plot(cx,cy)
axarr[0].set_title("The map")

# Zoomed plot of the final map (equal axis)
axarr[1].plot(nxf,nyf,'.')
axarr[1].plot(cx,cy)
axarr[1].set_xlim([cx[-1] - 0.1, cx[-1] +0.1])
axarr[1].set_ylim([cy[-1] - 0.1, cy[-1] +0.1])
axarr[1].set_title("Zoom")

# Zoomed plot of the final map (unequal axis)
axarr[2].plot(nxf,nyf,'.')
axarr[2].plot(cx,cy)
axarr[2].set_xlim([cx[-1] - 0.007, cx[-1] + 0.007])
axarr[2].set_ylim([cy[-1] - 0.007, cy[-1] + 0.007])
axarr[2].set_title("Stretch")
#axarr[1].set_xlim([cx[-1] - 0.1, cx[-1] + 0.1])
#axarr[1].set_ylim([cy[-1] - 0.1, cy[-1] + 0.1])
```

[11]:
```
Text(0.5, 1.0, 'Stretch')
```



How much faster is now to evaluate the Map rather than perform a new numerical integration?

[12]:
```
# First we profile the method evaluate (note that you need to call the method 4 times␣
↪to get the full state)
```

[13]:
```
%timeit xf.evaluate({"dr":epsilon, "dt":epsilon, "dvr":epsilon,"dvt":epsilon})
```

```
59.5 µs ± 4.74 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

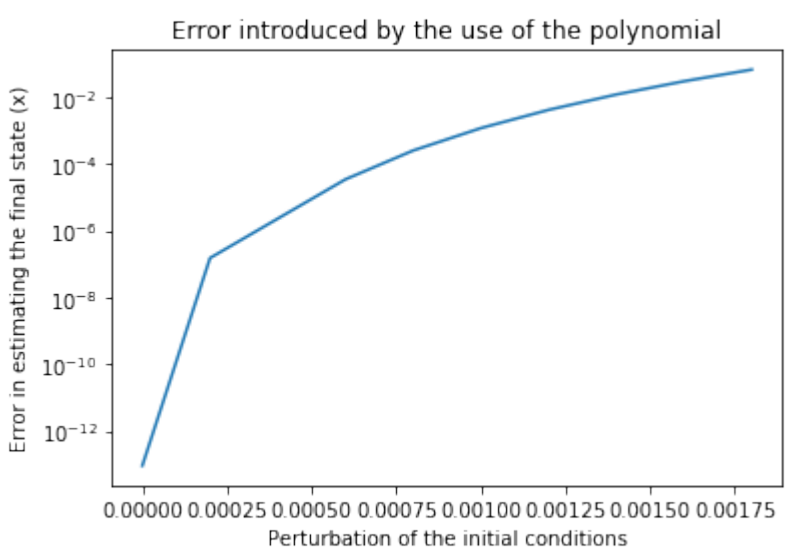
[14]:
```
# Then we profile the Runge-Kutta 4 integrator
```

[15]:
```
%%timeit
D.set_float_fmt('float64')
float_integration = de.OdeSystem(eom_kep_polar, y0=[it + epsilon for it in ic], dense_
↪output=False, t=(0, 300.), dt=0.01, rtol=1e-10, atol=1e-10, constants=dict())

float_integration.set_method("RK45")
float_integration.integrate(eta=False)
```

```
422 ms ± 31.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[16]:
```
# It seems the speedup is 2-3 orders of magnitude, but did we loose precision?
# We plot the error in the final result as computed by the HOTM and by the Runge-Kutta
# as a function of the distance from the original initial conditions
out = []
pert = np.arange(0,2e-3,2*1e-4)
for epsilon in pert:
    res_map_xf = xf.evaluate({"dr":epsilon, "dt":epsilon, "dvr":epsilon,"dvt":epsilon}
↪)
    res_int = de.OdeSystem(eom_kep_polar, y0=[it + epsilon for it in ic], dense_
↪output=False, t=(0, 300.), dt=0.01, rtol=1e-10, atol=1e-10, constants=dict())
    res_int.set_method("RK45")
    res_int.integrate()
    res_int_x = [it.y[0]*np.sin(it.y[2]) for it in res_int]
    res_int_xf = res_int_x[-1]
    out.append(np.abs(res_map_xf - res_int_xf))
plt.semilogy(pert,out)
plt.title("Error introduced by the use of the polynomial")
plt.xlabel("Perturbation of the initial conditions")
plt.ylabel("Error in estimating the final state (x)")
```

[16]:
```
Text(0, 0.5, 'Error in estimating the final state (x)')
```



6.6 Using pytorch

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

F

G

H

I

J

M

N

O

R

S

T

U

W

Y