simframe

Sebastian Stammler & Til Birnstiel

Feb 15, 2021

CONTENTS:

`simframe` is a framework to flexibly build scientific simulations.
It comes with a variety of integration schemes and some infrastructure for writing and reading data files.

To install `simframe` simply type
`pip install simframe`

Please have a look at the following examples to learn how to use `simframe`.

In this tutorial you'll learn how to

- set up frames
- create fields
- create integration variables
- set up derivatives
- set up integration instructions
- run simulations
- read and write data

For this notebook you need `matplotlib` and `scipy` in addition to the `simframe` requirements.

CHAPTER

ONE

SIMPLE INTEGRATION

In this tutorial we want to solve the most simple differential equation

$\frac{\mathrm{d}Y}{\mathrm{d}x} = b\,Y$

with the initial condition

$Y\,(0) = A.$

This problem has the solution

$Y\,(x) = A\,e^{bx}.$

We set up some parameters that allow us to easily change the problem and rerun the simulation.

[1]:
```
A  = 10.  # Initial value of Y
b  = -1.  # decay factor
dx =  0.1 # Stepsize
```

1.1 Setting up frames

Frame objects are the core functionality of `simframe`. They contain everything you need to run a simulation, from variable to parameters to integration schemes.

[2]:
```
from simframe import Frame
```

Here we set up a `Frame` object called `sim` and give it a meaningful description of our problem.

[3]:
```
sim = Frame(description="Simple Integration")
```

Right now the frame is empty. It contains attributes for integration and writing of data, that we need to specify later.

[4]:
```
sim
```

[4]:
```
Frame (Simple Integration)
--------------------------
    Integrator   : not specified
    Writer       : not specified
```

1.2 Creating fields

We can now fill the empty frame with our problem. First, we create a field for our variable `Y` and initialize it with its initial value `A`. Upon initialization field need to have the correct shape and data type already. This cannot be changed later.

[5]:
```
sim.addfield("Y", A)
```

The frame object has now the field `Y`.

[6]:
```
sim
```

[6]:
```
Frame (Simple Integration)
--------------------------
    Y            : Field
  -----
    Integrator   : not specified
    Writer       : not specified
```

You can do all kinds of operations with `Y` just as with `numpy.ndarray`.

[7]:
```
sim.Y + 3
```

[7]:
```
13.0
```

[8]:
```
import numpy as np
```

[9]:
```
np.exp(sim.Y)
```

[9]:
```
22026.465794806718
```

1.3 Setting up derivatives

To solve for `Y` we have to specify a derivative of the field that can be used by the integrator. The function for the derivative of any variable needs the frame object, the integration variable, and the variable itself as positional arguments and needs to return the value of the derivative.

`derivative(frame, x, Y)`

In our case here the derivative is very simple but more complex equations could also use different fields by adressing them via the `frame` object.

[10]:
```
def dYdx(frame, x, Y):
    return b*Y
```

Now we have to assign this function to the differentiator of our variable `Y`.

[11]:
```
sim.Y.differentiator = dYdx
```

The derivative can be called with `.derivative(x, Y)`. If you don't give `x` or `Y`, then `simframe` assumes the current values, which does not work at this moment, because we have not set `x`, yet.

[12]:
```
sim.Y.derivative(0., A)
```

[12]:
```
-10.0
```

1.4 Creating integration variables

Every frame objects needs at least one integration variable that controls the workflow and is advancing the simulation in space or time for example. In our case this is `x`. Integration variables can be added to the frame object just as fields.

[13]:
```
sim.addintegrationvariable("x", 0.)
```

The frame objects has now the integration variable `x`.

[14]:
```
sim
```

[14]:
```
Frame (Simple Integration)
--------------------------
    x             : Field, Integration variable
    Y             : Field
  -----
    Integrator    : not specified
    Writer        : not specified
```

The integration variable is used to advance the simulation in `x` in our case and needs to know about the stesize. We therefore have to create a function that returns `dx`. The only argument of this function has to be the frame object and needs to return the step size.

In our simple case we just want to return a constant step size that we defined earlier.

[15]:
```
def f_dx(frame):
    return dx
```

We now have to tell the updater of the itnegration variable to use this function.

[16]:
```
sim.x.updater = f_dx
```

In addition to that, the integration variable needs to know about snapshots, i.e. points in space or time, when data should be written. Even you don't want to write data, you need to give at least one fineal value, because `simframe` needs to know when to stop the calculation. The snapshots have to be either a list or an array with the desired snapshots in increasing order.

[17]:
```
sim.x.snapshots = np.linspace(1., 10., 10)
```

Compared to regular fields, integration variables have additional functionality. For example we can get the current step size, the maximum possible step size until the next snapshot is written, and the value of the integration variable at the next snapshot.

[18]:
```
sim.x.stepsize
```

[18]:
```
0.1
```

[19]:
```
sim.x.maxstepsize
```

[19]:
```
1.0
```

[20]:
```
sim.x.nextsnapshot
```

[20]:
```
1.0
```

The previously taken stepsize can be accessed with the following attribute, which is set to 0 by default upon initialization.

[21]:
```
sim.x.prevstepsize
```

[21]:
```
0.0
```

1.5 Setting up the integrator

So far we have set up our variables. But we also need to set up an integrator that is performing the actual integration. In our case we want to use a simple explicit Euler 1st-order scheme. The integrator needs the integration variable as positional argument during initialization. We could therefore for example set up different integrators with different integration variables and exchange them midway.

[22]:
```
from simframe import Integrator
```

[23]:
```
sim.integrator = Integrator(sim.x, description="Euler 1st-order")
```

The frame object has now an integrator set.

[24]:
```
sim
```

[24]:
```
Frame (Simple Integration)
--------------------------
    x             : Field, Integration variable
    Y             : Field
  -----
    Integrator    : Integrator (Euler 1st-order)
    Writer        : not specified
```

The integrator is basically a container for integration instructions. We therefore have to give tell it which integration instructions it should perform. Instructions have to be a list of `Instruction` objects, which need an integration scheme and a field to be integrated as positional arguments.

[25]:
```
from simframe import Instruction
from simframe import schemes
```

[26]:
```
sim.integrator.instructions = [Instruction(schemes.expl_1_euler, sim.Y)]
```

1.6 Setting up writers

The simulation is now basically ready to go. But in this example we also want to write data files. We therefore have to specify a writer to our frame object. In this case we want to write data files in the `hdf5` format.

[27]:
```
from simframe import writers
```

[28]:
```
sim.writer = writers.hdf5writer
```

The `hdf5writer` come with a few pre-defined options.

[29]:
```
sim.writer
```

[29]:
```
Writer (HDF5 file format using h5py)
------------------------------------
```

(continues on next page)

(continued from previous page)

```
    Data directory : data
    File names     : data/data0000.hdf5
    Overwrite      : False
    Dumping        : True
    Options        : {'com': 'lzf', 'comopts': None}
    Verbosity      : 1
```

First, we want to change the data directory to which the files are written. If the data directory does not exist, `simframe` will create it.

[30]:
```
sim.writer.datadir = "1_data"
```

And second, we want the writer to overwrite existing files. In that way we can easily restart the notebook with different parameters. Usually a writer will not allow you to overwrite existing files to protect your data.

[31]:
```
sim.writer.overwrite = True
```

[32]:
```
sim.writer
```

[32]:
```
Writer (HDF5 file format using h5py)
------------------------------------
    Data directory : 1_data
    File names     : 1_data/data0000.hdf5
    Overwrite      : True
    Dumping        : True
    Options        : {'com': 'lzf', 'comopts': None}
    Verbosity      : 1
```

By default the writer is writing dump files at every snapshot. In contrast to data files, which only contain the data, a dump file contains the entire frame object from which the simulation can be restarted if anything went wrong. Since these files can be large for bigger projects, it always overwrites the existing dump file.

1.7 Running simulations

The frame objects is now completely set up and we are ready to go.

[33]:
```
sim.run()
```

```
Creating data directory '1_data'.
Writing file 1_data/data0000.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0001.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0002.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0003.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0004.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0005.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0006.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0007.hdf5
```

(continues on next page)

(continued from previous page)

```
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0008.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0009.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0010.hdf5
Writing dump file 1_data/frame.dmp
Execution time: 0:00:00
```

1.8 Reading data

Every writer should come with a reader that contains instructions for reading the data files the writer has written. It is possible to read a single output file.

[34]:
```
data3 = sim.writer.read.output(3)
```

This returns the namespace containing the data of the chosen output file.

[35]:
```
data3
```

[35]:
```
namespace(Y=array([0.42391158]),
          description='Simple Integration',
          x=array([3.]))
```

You can check for existing files in the data directory.

[36]:
```
sim.writer.read.listfiles()
```

[36]:
```
['1_data/data0000.hdf5',
 '1_data/data0001.hdf5',
 '1_data/data0002.hdf5',
 '1_data/data0003.hdf5',
 '1_data/data0004.hdf5',
 '1_data/data0005.hdf5',
 '1_data/data0006.hdf5',
 '1_data/data0007.hdf5',
 '1_data/data0008.hdf5',
 '1_data/data0009.hdf5',
 '1_data/data0010.hdf5']
```

Or you can read the complete data that is in the data directory.

[37]:
```
data = sim.writer.read.all()
```

The fields can be easily adressed just as with the frame object.

[38]:
```
data.Y
```

[38]:
```
array([1.00000000e+01, 3.48678440e+00, 1.21576655e+00, 4.23911583e-01,
       1.47808829e-01, 5.15377521e-02, 1.79701030e-02, 6.26578748e-03,
       2.18474501e-03, 7.61773480e-04, 2.65613989e-04])
```

Instead of reading the full data set or a single snapshots, it is also possible to read a single field from all snapshots.

[39]:
```
seq = sim.writer.read.sequence("x")
```

[40]:
```
seq
```

[40]:
```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

We now want to fit a function to the data to see, if we can reconstruct the initial parameters.

[41]:
```
def fit(x, A, b):
    return A*np.exp(b*x)
```

[42]:
```
from scipy.optimize import curve_fit
```

[43]:
```
popt, pcov = curve_fit(fit, data.x, data.Y)
```

[44]:
```
from IPython.display import Markdown as md
md("| |Simulation|Analytical Solution|\n|:-:|:-:|:-:|\n|A|{:4.2f}|{:4.2f}|\n|b|{:4.2f}
↪|{:4.2f}|".format(popt[0],A,popt[1],b))
```

[44]:

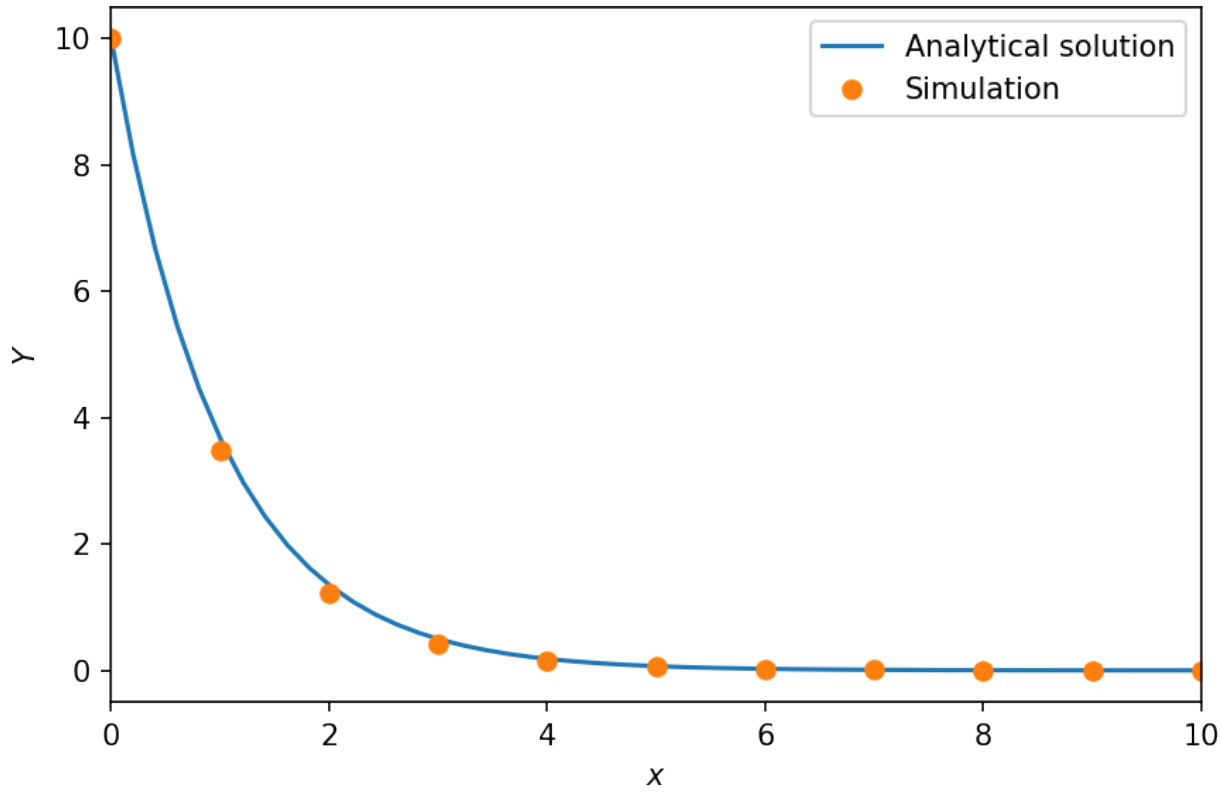
	Simulation	Analytical Solution
A	10.00	10.00
b	-1.05	-1.00

Additionally we want to plot our data.

[45]:
```
import matplotlib.pyplot as plt
```

[46]:
```
def plot(data):
    fig, ax = plt.subplots(dpi=150)
    x = np.linspace(0., 20., 100)
    ax.plot(x, fit(x, A, b), label="Analytical solution")
    ax.plot(data.x, data.Y, "o", label="Simulation")
    ax.set_xlim(data.x[0], data.x[-1])
    ax.set_xlabel("$x$")
    ax.set_ylabel("$Y$")
    ax.legend()
    fig.tight_layout()
```

[47]:
```
plot(data)
```



1.9 Reading dump files

Let's say we want to continue the simulation from a dump file that we have stored somewhere. We first have to read the file with `readdump(filename)` which needs the path to the file as argument and which returns a frame object.

[48]:
```
from simframe.io import readdump
```

[49]:
```
sim_cont = readdump("1_data/frame.dmp")
```

[50]:
```
sim_cont
```

[50]:
```
Frame (Simple Integration)
--------------------------
    x            : Field, Integration variable
    Y            : Field
  -----
    Integrator   : Integrator (Euler 1st-order)
    Writer       : Writer (HDF5 file format using h5py)
```

We only have to add a few more snapshots.

[51]:
```
sim_cont.x.snapshots = np.linspace(1., 20., 20)
```

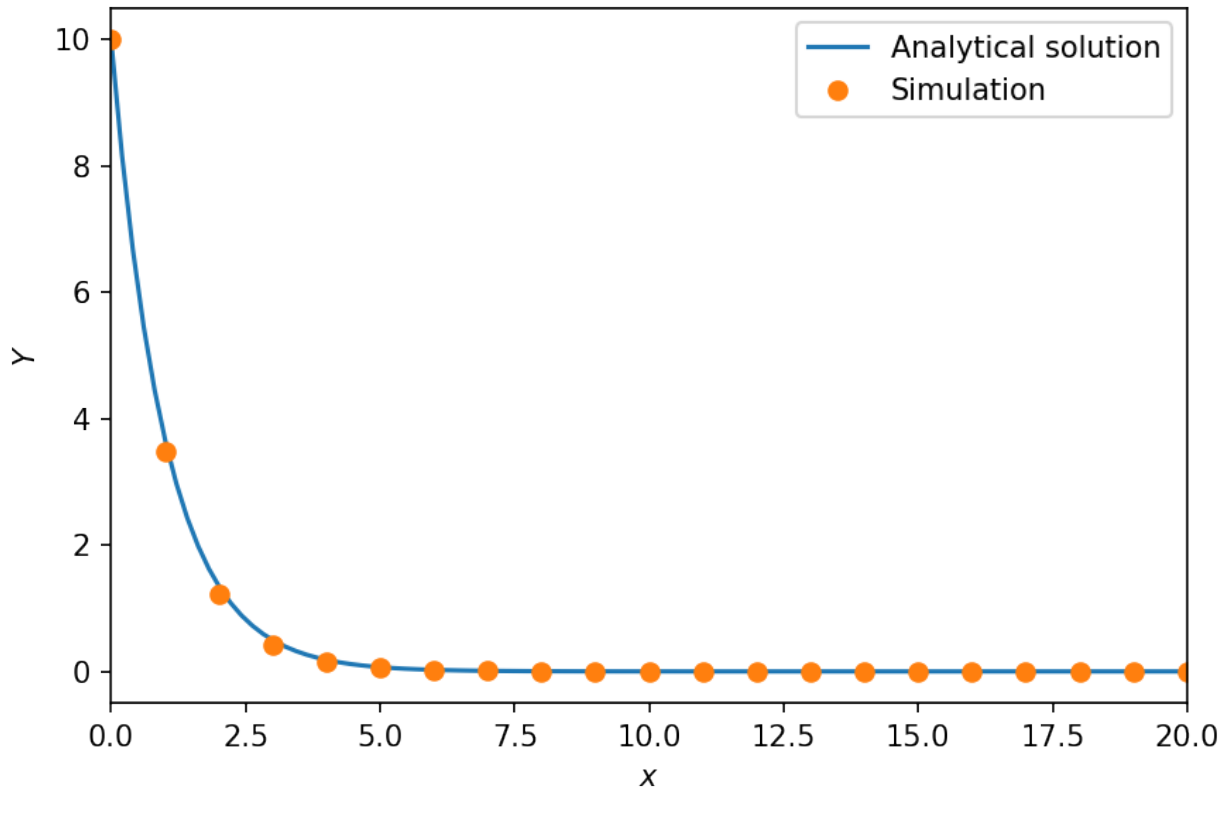
Now we can restart the simulation from the current state, read the data, and plot it.

[52]:
```
sim_cont.run()
```

```
Writing file 1_data/data0011.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0012.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0013.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0014.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0015.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0016.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0017.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0018.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0019.hdf5
Writing dump file 1_data/frame.dmp
Writing file 1_data/data0020.hdf5
Writing dump file 1_data/frame.dmp
Execution time: 0:00:00
```

[53]:
```
data = sim_cont.writer.read.all()
```

[54]:
```
plot(data)
```



1.10 Progress bar

If you are running `simframe` on an interactive shell if will show by default a progress bar with an estimate of the expected computation time. If you want to turn it off to save some computational overhead you can do so by reducing the verbosity of the frame object.

[55]:
```
sim.verbosity = 0
```

In this tutorial you'll learn how to

- create groups
- create immutable fields
- display the entire content tree
- set up complex instruction sets
- set up leapfrog integration

For this notebook you need `astropy` and `matplotlib` in addition to the `simframe` requirements.

CHAPTER

TWO

ADVANCED INTEGRATION

Let's have a look at orbital integration.
Consider a planet Earth orbiting the Sun. How does its orbit look like?

The equations of motion are determined by a set of two differential equations for the position $\vec{r}$ and velocity $\vec{v}$ of Earth and Sun..

$\frac{\mathrm{d}}{\mathrm{d}t}\vec{r} = \vec{v}$

$m\frac{\mathrm{d}}{\mathrm{d}t}\vec{v} = \vec{F}_\mathrm{G}$

The gravitational force F_G of a body of mass M and position $\vec{R}$ acting on a body of mass m at position $\vec{r}$ is given by

$\vec{F}_\mathrm{G} = -GmM\frac{\vec{r}-\vec{R}}{|\vec{r}-\vec{R}|^3}$

2.1 Creating groups

First we need to add fields for postion, velocity, and mass for Earth and Sun.
To have more structure we group the fields by Earth and Sun. This can be done with `addgroup(name)`.

[1]:
```
from simframe import Frame
```

[2]:
```
sim = Frame(description="Earth-Sun system")
```

[3]:
```
sim.addgroup("Sun")
sim.addgroup("Earth")
```

The frame object has now two groups for Earth and Sun, that can be adressed just as fields.

[4]:
```
sim
```

[4]:
```
Frame (Earth-Sun system)
------------------------
    Earth          : Group
    Sun            : Group
  -----
    Integrator     : not specified
    Writer         : not specified
```

We will keep the problem general in three dimensions, but we will only use two of them for simplicity.
Although spherical coordinates would be more suitable for the problem, we will use Carteesian coordinates here.

We'll use the `astropy.constants` and `astropy.units` modules for the constants and units we need here.

[5]:
```
import astropy.constants as c
import astropy.units as u
```

[6]:
```
AU    = (1.*u.au).si.value     # Astronomical unit [m]
day   = (1.*u.day).si.value    # Day [s]
G     = c.G.si.value           # Gravitational constant [m^3/kg/s^2]
year  = (1.*u.year).si.value   # Year [s]

M_earth = c.M_earth.si.value # Mass of the Earth [kg]
M_sun   = c.M_sun.si.value   # Mass of the Sun [kg]
```

2.2 Creating immutable fields

We can now fill the groups with the variables we need, starting with the masses.
Again: the initial values must have the correct shape and data types.

[7]:
```
sim.Earth.addfield("M", M_earth, description="Mass [kg]")
```

The groups and the fields within can be easily accessed with

[8]:
```
sim.Earth
```

[8]:
```
Group
-----
    M             : Field (Mass [kg])
  -----
```

[9]:
```
sim.Earth.M
```

[9]:
```
5.972167867791379e+24
```

The mass of the Earth shall be constant throughout the simulation. We therefore set a flag so we cannot accidentially change its value.

[10]:
```
sim.Earth.M.constant = True
```

[11]:
```
sim.Earth
```

[11]:
```
Group
-----
    M             : Field (Mass [kg]), constant
  -----
```

Now we add the mass of the Sun, which we directly set to constant when adding the field.

[12]:
```
sim.Sun.addfield("M", M_sun, constant=True, description="Mass [kg]")
```

[13]:
```
sim.Sun
```

[13]:
```
Group
-----
    M             : Field (Mass [kg]), constant
  -----
```

We can do all sorts of operations with those fields, such as calculating the mass ratio.

[14]:
```
sim.Earth.M / sim.Sun.M
```

[14]:
```
3.0034893488507934e-06
```

Now we need to add fields for position and velocity. We initialize them with zeros and think later about the initial conditions.
The important thing is that the have the right shape upon initialization.

[15]:
```
import numpy as np
```

[16]:
```
sim.Earth.addfield("r", np.zeros(3), description="Position [m]")
sim.Earth.addfield("v", np.zeros(3), description="Velocity [m/s]")
sim.Sun.addfield("r", np.zeros(3), description="Position [m]")
sim.Sun.addfield("v", np.zeros(3), description="Velocity [m/s]")
```

[17]:
```
sim.Earth
```

[17]:
```
Group
-----
    M             : Field (Mass [kg]), constant
    r             : Field (Position [m])
    v             : Field (Velocity [m/s])
  -----
```

[18]:
```
sim.Sun
```

[18]:
```
Group
-----
    M             : Field (Mass [kg]), constant
    r             : Field (Position [m])
    v             : Field (Velocity [m/s])
  -----
```

Setting the integration variable

For our simulation we need an integration variable. In our case this is the time.

[19]:
```
sim.addintegrationvariable("t", 0., description="Time [s]")
```

We'll set the step size to a constant value of one day.

[20]:
```
dt = 1.*day
```

[21]:
```
def f_dt(frame):
    return dt
```

[22]:
```
sim.t.updater = f_dt
```

We want to integrate for two years and want to have a snapshot every ten days.

[23]:
```
snapwidth = 10.*day
tmax = 2.*year
```

[24]:
```
sim.t.snapshots = np.arange(snapwidth, tmax, snapwidth)
```

Note: If the initial value of the integration variable is smaller the first snapshot, `simframe` automatically writes an output with initial conditions, if a writer is set.

[25]:
```
sim
```

[25]:
```
Frame (Earth-Sun system)
------------------------
    Earth         : Group
    Sun           : Group
  -----
    t             : Field (Time [s]), Integration variable
  -----
    Integrator    : not specified
    Writer        : not specified
```

Setting the writer

As a writer we use the namespacewriter, which does not write the data into files, but stores them within a buffer in the writer.

[26]:
```
from simframe import writers
```

[27]:
```
sim.writer = writers.namespacewriter
```

We do not want to write dump files here.

[28]:
```
sim.writer.dumping = False
```

[29]:
```
sim.writer
```

[29]:
```
Writer (Namespace writer)
-------------------------
    Data directory : data
    Dumping        : False
    Verbosity      : 1
```

Adding differential equations

As a next step we'll add differential equations to the quantities. The differential equations for the positions are simple. We simply return the velocities, which we can adress with the `frame` argument.

[30]:
```
def dr_Earth(frame, x, Y):
    return frame.Earth.v

def dr_Sun(frame, x, Y):
    return frame.Sun.v
```

For the differential equations of the velocities we'll write a little helper function that computes the gravitational acceleration.

[31]:
```
# Gravitational acceleration
def ag(M, r, R):
    direction = r-R
    distance = np.linalg.norm(direction)
    return -G * M * direction / distance**3
```

[32]:
```
def dv_Earth(frame, x, Y):
    return ag(frame.Sun.M, frame.Earth.r, frame.Sun.r)

def dv_Sun(frame, x, Y):
    return ag(frame.Earth.M, frame.Sun.r, frame.Earth.r)
```

Now we need to add the differential equations to their fields.

[33]:
```
sim.Earth.v.differentiator = dv_Earth
sim.Earth.r.differentiator = dr_Earth
sim.Sun.v.differentiator   = dv_Sun
sim.Sun.r.differentiator   = dr_Sun
```

2.3 Displaying the table of contents

You can also display the complete tree structure of your frame.

[34]:
```
sim.toc
```

```
Frame (Earth-Sun system)
    - Earth: Group
        - M: Field (Mass [kg]), constant
        - r: Field (Position [m])
        - v: Field (Velocity [m/s])
    - Sun: Group
        - M: Field (Mass [kg]), constant
        - r: Field (Position [m])
        - v: Field (Velocity [m/s])
    - t: Field (Time [s]), Integration variable
```

2.4 Setting up complex integration instructions

Next we need to set up the integrator. We integrate all quantities with the explicit Euler 1st-order scheme as in the previous tutorial, but this time he have to integrate four fields. This can be easily achieved by adding four instructions.

[35]:
```
from simframe import Integrator
from simframe.integration import Instruction
from simframe import schemes
```

[36]:
```
sim.integrator = Integrator(sim.t, description="Euler 1st-order")
```

[37]:
```
instructions_euler = [Instruction(schemes.expl_1_euler, sim.Earth.r),
                      Instruction(schemes.expl_1_euler, sim.Earth.v),
                      Instruction(schemes.expl_1_euler, sim.Sun.r  ),
                      Instruction(schemes.expl_1_euler, sim.Sun.v  ),
                     ]
```

[38]:
```
sim.integrator.instructions = instructions_euler
```

Initial conditions

Before we can start the simulation, we have to think about initial conditions.

If we simply set the Sun at rest in the center of our simulation and only set the position and velocity of the Earth, then the center of mass would have a non-zero momentum and would slowly drift away from its initial position.

So what we do first: we set the Sun's positon to zero (i.e., don't do anything) and the Earth's position to a distance of 1 AU in positive x-direction. Then we'll offset their positions to center the system on the center of mass instead onto the Sun.

[39]:
```
r_Earth_ini = np.array([AU, 0., 0.])
r_Sun_ini = np.zeros(3)
# Center of mass
COM_ini = (M_earth*r_Earth_ini + M_sun*r_Sun_ini) / (M_earth+M_sun)
# Offset both positions
r_Earth_ini -= COM_ini
r_Sun_ini -= COM_ini
```

We save them in a separate variable instead of assigning them directly for later use.

The inital orbital velocities of the Earth shall be in positive y-direction, the velocity of the Sun in negative y-direction. For calculating the value we use the reduced mass mu of the system.

[40]:
```
mu = M_earth*M_sun / (M_earth+M_sun)
```

[41]:
```
v_Earth_ini = np.array([0., np.sqrt(G*M_sun/M_earth*mu/AU), 0.])
v_Sun_ini   = np.array([0., -np.sqrt(G*M_earth/M_sun*mu/AU), 0.])
```

Now we assign them to their fields.

[42]:
```
sim.Earth.r = r_Earth_ini
sim.Earth.v = v_Earth_ini
sim.Sun.r   = r_Sun_ini
sim.Sun.v   = v_Sun_ini
```

Starting the simulation

[43]:
```
sim.run()
```

```
Saving frame 0
Saving frame 1
Saving frame 2
Saving frame 3
Saving frame 4
Saving frame 5
Saving frame 6
Saving frame 7
Saving frame 8
Saving frame 9
Saving frame 10
Saving frame 11
Saving frame 12
```

(continues on next page)

(continued from previous page)

```
Saving frame 13
Saving frame 14
Saving frame 15
Saving frame 16
Saving frame 17
Saving frame 18
Saving frame 19
Saving frame 20
Saving frame 21
Saving frame 22
Saving frame 23
Saving frame 24
Saving frame 25
Saving frame 26
Saving frame 27
Saving frame 28
Saving frame 29
Saving frame 30
Saving frame 31
Saving frame 32
Saving frame 33
Saving frame 34
Saving frame 35
Saving frame 36
Saving frame 37
Saving frame 38
Saving frame 39
Saving frame 40
Saving frame 41
Saving frame 42
Saving frame 43
Saving frame 44
Saving frame 45
Saving frame 46
Saving frame 47
Saving frame 48
Saving frame 49
Saving frame 50
Saving frame 51
Saving frame 52
Saving frame 53
Saving frame 54
Saving frame 55
Saving frame 56
Saving frame 57
Saving frame 58
Saving frame 59
Saving frame 60
Saving frame 61
Saving frame 62
Saving frame 63
Saving frame 64
Saving frame 65
Saving frame 66
Saving frame 67
Saving frame 68
Saving frame 69
```

(continues on next page)

(continued from previous page)

```
Saving frame 70
Saving frame 71
Saving frame 72
Saving frame 73
Execution time: 0:00:01
```

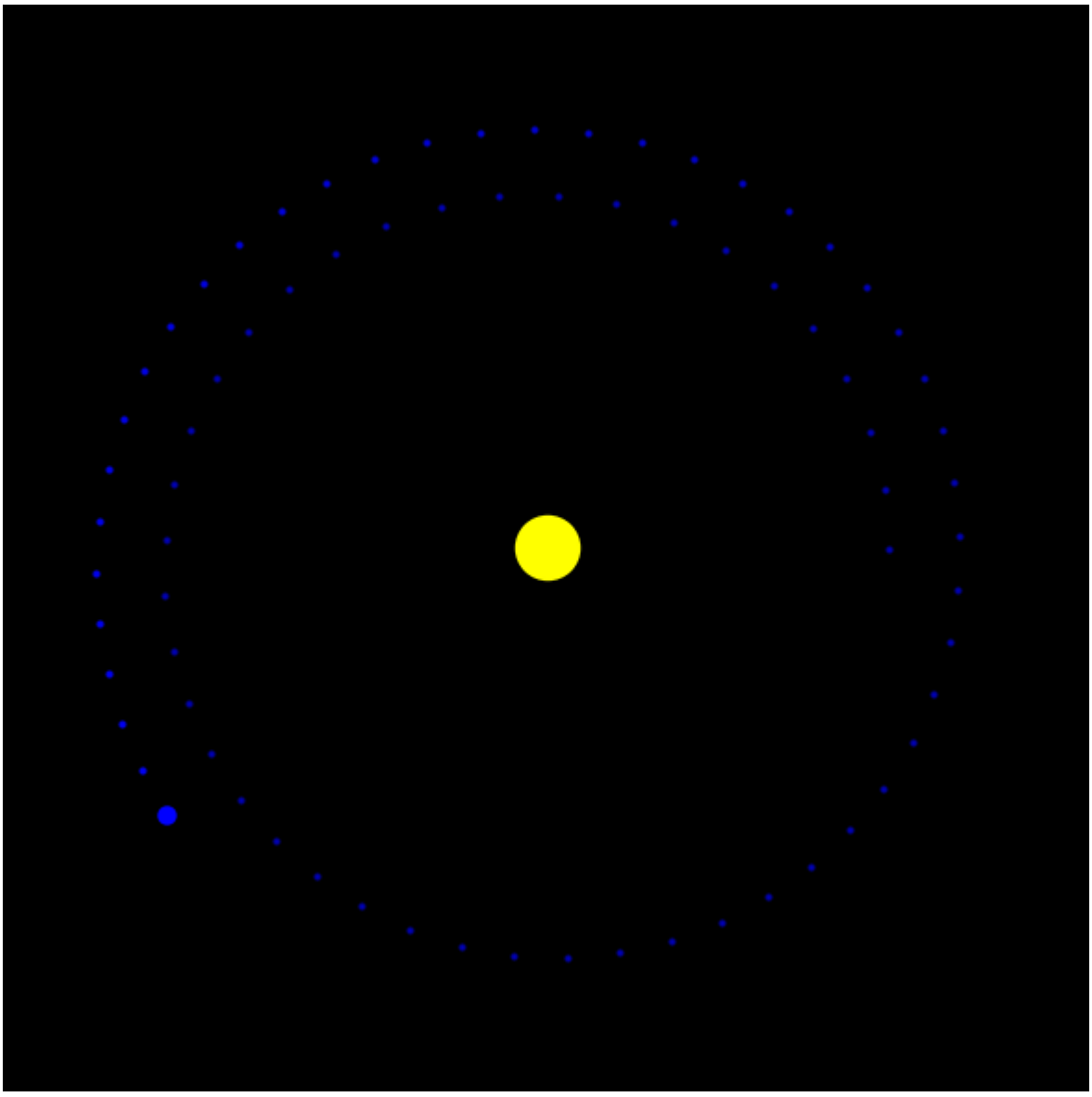
Reading and plotting

Reading data from the namespace writer works identical to the writer discussed earlier.

[44]:
```
data_euler = sim.writer.read.all()
```

[45]:
```
import matplotlib.pyplot as plt

def plot_orbits(data):
    fig, ax = plt.subplots(dpi=150)
    ax.set_aspect(1)
    ax.axis("off")
    fig.set_facecolor("#000000")
    imax = data.t.shape[0]
    for i in range(imax):
        alpha = np.maximum(i/imax-0.1, 0.5)
        ax.plot(data.Sun.r[i, 0], data.Sun.r[i, 1], "o", c="#FFFF00", markersize=4,␣
↪alpha=alpha)
        ax.plot(data.Earth.r[i, 0], data.Earth.r[i, 1], "o", c="#0000FF",␣
↪markersize=1, alpha=alpha)
    ax.plot(data.Sun.r[-1, 0], data.Sun.r[-1, 1], "o", c="#FFFF00", markersize=16)
    ax.plot(data.Earth.r[-1, 0], data.Earth.r[-1, 1], "o", c="#0000FF", markersize=4)
    ax.set_xlim(-1.5*AU, 1.5*AU)
    ax.set_ylim(-1.5*AU, 1.5*AU)
    fig.tight_layout()
```

[46]:
```
plot_orbits(data_euler)
```



2.5 Leapfrog integration

As you can see, the Earth is not on a circular orbit. Its orbital distance is increasing and because of that the Earth could not achieve two full orbital cycles.

The problem is the simple integration scheme used here.
Every integration scheme has numerical error. Some have larger ones than others. Euler's 1st order method is simply not suited for orbital integration.

But there is a way out: Symplectic integration
Symplectic integrators conserve the energy of the system.

One of these methods is the Leapfrog method.
Leapfrogging means the velocity and the position are not updated synchronous, but in between each other. They are leapfroggin each other.

In our case we first update the velocities by using half of the step size. Then we update the positions for a full time step. And finally, we update the velocities for another semi time step. We can do this by using the `fstep` keyword argument for the instructions, which is the fraction of the time for which this instruction is applied.

But there is one caveat: usually the fields that are integrated in an instructions set are only updated after all instructions have been executed, such that the second instruction still sees the original value of the first variable and not already the new value.

But for leapfrog integration, the instruction for the positions already need to know the new values of the velocities. And the second set of velocity instructions already need to know the new positions. We therefore have to add `update` instructions in between.

We do not have to add update instructions after the second set of velocity operations, because at the end of an instruction set all fields contained in the instruction set will be updated.

The new leapfrog instruction set now looks like this.

[47]:
```
instructions_leapfrog = [Instruction(schemes.expl_1_euler, sim.Sun.v,   fstep=0.5),
                         Instruction(schemes.expl_1_euler, sim.Earth.v, fstep=0.5),
                         Instruction(schemes.update,       sim.Sun.v             ),
                         Instruction(schemes.update,       sim.Earth.v           ),
                         Instruction(schemes.expl_1_euler, sim.Sun.r,   fstep=1.0),
                         Instruction(schemes.expl_1_euler, sim.Earth.r, fstep=1.0),
                         Instruction(schemes.update,       sim.Sun.r             ),
                         Instruction(schemes.update,       sim.Earth.r           ),
                         Instruction(schemes.expl_1_euler, sim.Sun.v,   fstep=0.5),
                         Instruction(schemes.expl_1_euler, sim.Earth.v, fstep=0.5),
                        ]
```

We can assign this new instruction set to our frame. To rerun the simulation we have to reset the initial conditions, that we've saved earlier.

[48]:
```
sim.integrator.instructions = instructions_leapfrog
sim.integrator.description = "Leapfrog integrator"
sim.Earth.r = r_Earth_ini
sim.Earth.v = v_Earth_ini
sim.Sun.r   = r_Sun_ini
sim.Sun.v   = v_Sun_ini
sim.t       = 0.
```

Additionally we have to reset the buffer of the namespace writer. Otherwise, we would simply add snapshots to the old dataset. Furthermore, we descrease the `verbosity` of the writer to prevent it from writing information on screen.

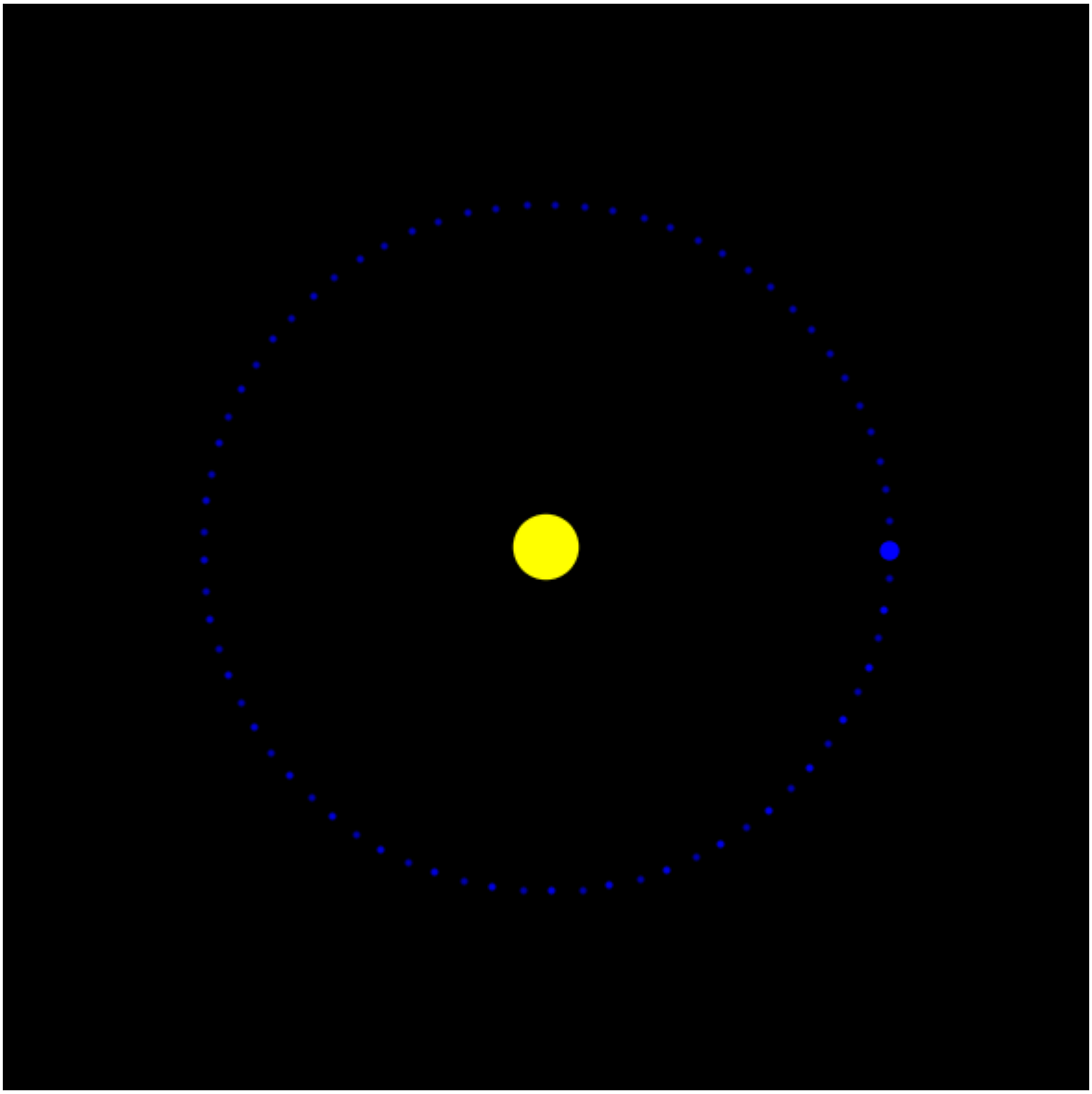
[49]:
```
sim.writer.reset()
sim.writer.verbosity = 0
```

[50]:
```
sim.run()
```

```
Execution time: 0:00:01
```

[51]:
```
data_leapfrog = sim.writer.read.all()
```

[52]:
```
plot_orbits(data_leapfrog)
```



In this tutorial you'll learn how to

- update groups and fields
- use heartbeats

For this notebook you need `astropy` and `matplotlib` in addition to the `simframe` requirements.

CHAPTER

THREE

UPDATING GROUPS AND FIELDS

In this example we revisit the orbital integration from the previous tutorial. But this time we also want to calculate the equilibrium temperature on Earth. That is not a quantity that we need to integrate. We can simply calculate it from the other quantities. But we have to tell `simframe` how to do it.

Setting up the problem

The problem setup follows the previous example. Please have a look there for more details.

[1]:
```
from simframe import Frame
```

[2]:
```
sim = Frame(description="Earth-Sun system")
```

[3]:
```
sim.addgroup("Sun")
sim.addgroup("Earth")
```

[4]:
```
import astropy.constants as c
import astropy.units as u
```

[5]:
```
AU   = (1.*u.au).si.value
day  = (1.*u.day).si.value
G    = c.G.si.value
year = (1.*u.year).si.value

M_earth = c.M_earth.si.value
M_sun   = c.M_sun.si.value
```

Earth

[6]:
```
import numpy as np
```

[7]:
```
sim.Earth.addfield("M", M_earth, description="Mass [kg]", constant=True)
sim.Earth.addfield("r", np.zeros(3), description="Position [m]")
sim.Earth.addfield("v", np.zeros(3), description="Velocity [m/s]")
```

Sun

[8]:
```
sim.Sun.addfield("M", c.M_sun.si.value, description="Mass [kg]", constant=True)
sim.Sun.addfield("r", np.zeros(3), description="Position [m]")
sim.Sun.addfield("v", np.zeros(3), description="Velocity [m/s]")
```

Integration variable

[9]:
```
sim.addintegrationvariable("t", 0., description="Time [s]")
```

[10]:
```
dt = 1.*day
```

[11]:
```
def f_dt(frame):
    return dt
```

[12]:
```
sim.t.updater = f_dt
```

[13]:
```
snapwidth = 5.*day
tmax = 2.*year
```

[14]:
```
sim.t.snapshots = np.arange(snapwidth, tmax+1, snapwidth)
```

Writer

[15]:
```
from simframe import writers
```

[16]:
```
sim.writer = writers.namespacewriter
```

[17]:
```
sim.writer.dumping = False
sim.writer.verbosity = 0
```

Differential equations

[18]:
```
def dr_Earth(frame, x, Y):
    return frame.Earth.v

def dr_Sun(frame, x, Y):
    return frame.Sun.v
```

[19]:
```
# Gravitational acceleration
def ag(M, r, R):
    direction = r-R
    distance = np.linalg.norm(direction)
    return -G * M * direction / distance**3
```

[20]:
```
def dv_Earth(frame, x, Y):
    return ag(frame.Sun.M, frame.Earth.r, frame.Sun.r)

def dv_Sun(frame, x, Y):
    return ag(frame.Earth.M, frame.Sun.r, frame.Earth.r)
```

[21]:
```
sim.Earth.v.differentiator = dv_Earth
sim.Earth.r.differentiator = dr_Earth
sim.Sun.v.differentiator   = dv_Sun
sim.Sun.r.differentiator   = dr_Sun
```

Integrator

[22]:
```
from simframe import Integrator
```

[23]:
```
integrator = Integrator(sim.t, description="Leapfrog integratior")
```

[24]:
```
from simframe import schemes
from simframe.integration import Instruction
```

Leapfrog integrator

[25]:
```
instructions = [Instruction(schemes.expl_1_euler, sim.Sun.v,   fstep=0.5),
                Instruction(schemes.expl_1_euler, sim.Earth.v, fstep=0.5),
                Instruction(schemes.update,       sim.Sun.v             ),
                Instruction(schemes.update,       sim.Earth.v           ),
                Instruction(schemes.expl_1_euler, sim.Sun.r,   fstep=1.0),
                Instruction(schemes.expl_1_euler, sim.Earth.r, fstep=1.0),
                Instruction(schemes.update,       sim.Sun.r             ),
                Instruction(schemes.update,       sim.Earth.r           ),
                Instruction(schemes.expl_1_euler, sim.Sun.v,   fstep=0.5),
                Instruction(schemes.expl_1_euler, sim.Earth.v, fstep=0.5),
               ]
```

[26]:
```
integrator.instructions = instructions
```

[27]:
```
sim.integrator = integrator
```

Initial conditions

In this example we want to send the Earth on an eccentric orbit to have a seasonal change in the equilibrium temperature. For this we simply incline the initial velocities of Earth an Sun by an angle `alpha`.

[28]:
```
r_Earth_ini = np.array([AU, 0., 0.])
r_Sun_ini = np.zeros(3)
# Center of mass
COM_ini = (M_earth*r_Earth_ini + M_sun*r_Sun_ini) / (M_earth+M_sun)
# Offset both positions
r_Earth_ini -= COM_ini
r_Sun_ini -= COM_ini
```

[29]:
```
mu = M_earth*M_sun / (M_earth+M_sun)
```

[30]:
```
alpha = 30. * u.deg
v_Earth_ini = np.sqrt(G*M_sun/M_earth*mu/AU)
v_Earth_ini = np.array([-np.sin(alpha)*v_Earth_ini, np.cos(alpha)*v_Earth_ini, 0.])
v_Sun_ini   = np.sqrt(G*M_earth/M_sun*mu/AU)
v_Sun_ini   = np.array([np.sin(alpha)*v_Sun_ini, -np.cos(alpha)*v_Sun_ini, 0.])
```

[31]:
```
sim.Earth.r = r_Earth_ini
sim.Earth.v = v_Earth_ini
sim.Sun.r   = r_Sun_ini
sim.Sun.v   = v_Sun_ini
```

Starting

[32]:
```
sim.run()
```

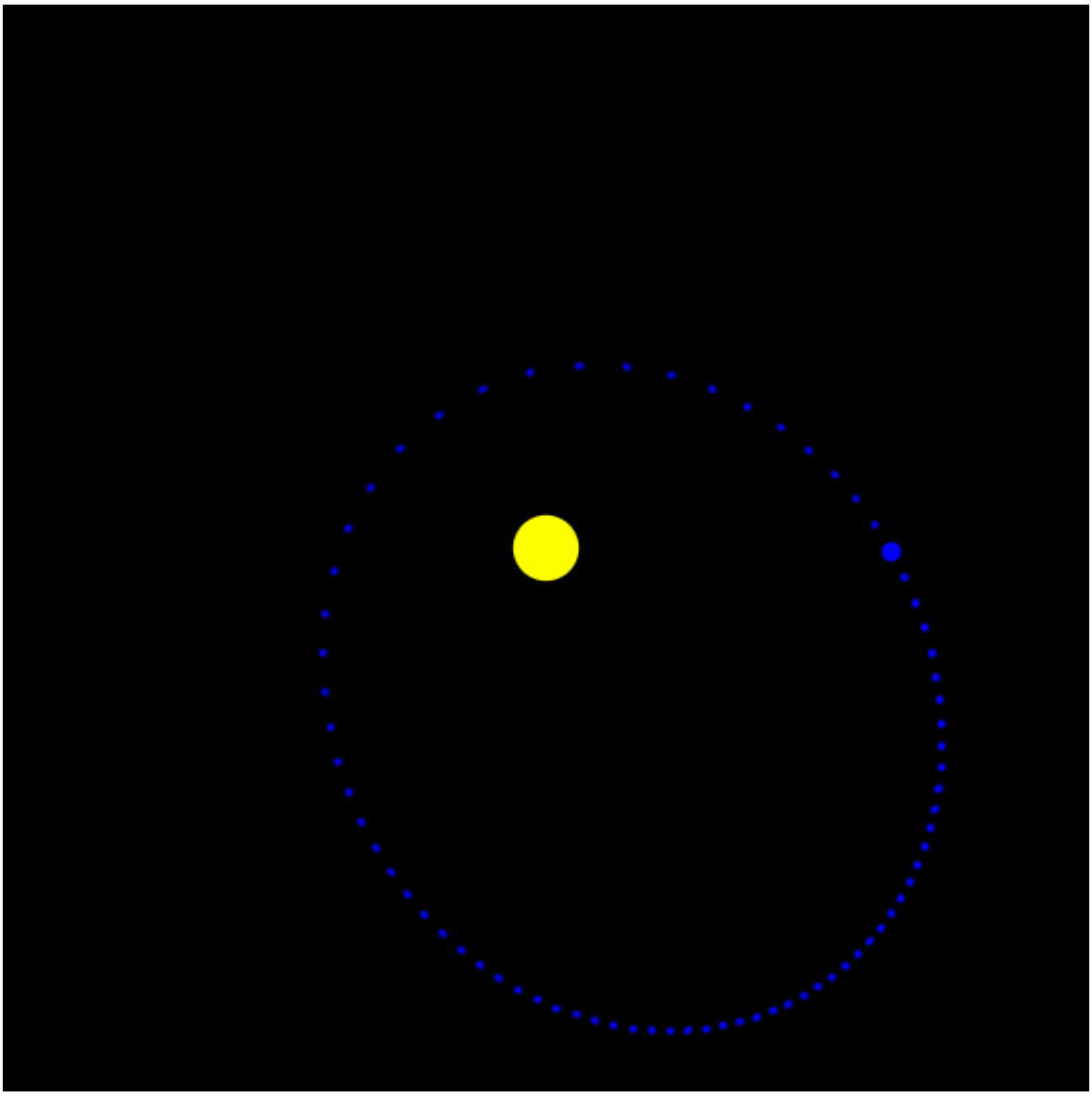
```
Execution time: 0:00:02
```

Reading and plotting

[33]:
```
data = sim.writer.read.all()
```

[34]:
```
import matplotlib.pyplot as plt

def plot_orbits(data):
    fig, ax = plt.subplots(dpi=150)
    ax.set_aspect(1)
    ax.axis("off")
    fig.set_facecolor("#000000")
    imax = data.t.shape[0]
    for i in range(imax):
        alpha = np.maximum(i/imax-0.1, 0.5)
        ax.plot(data.Sun.r[i, 0], data.Sun.r[i, 1], "o", c="#FFFF00", markersize=4,␣
↪alpha=alpha)
        ax.plot(data.Earth.r[i, 0], data.Earth.r[i, 1], "o", c="#0000FF",␣
↪markersize=1, alpha=alpha)
    ax.plot(data.Sun.r[-1, 0], data.Sun.r[-1, 1], "o", c="#FFFF00", markersize=16)
    ax.plot(data.Earth.r[-1, 0], data.Earth.r[-1, 1], "o", c="#0000FF", markersize=4)
    ax.set_xlim(-1.5*AU, 1.5*AU)
    ax.set_ylim(-1.5*AU, 1.5*AU)
    fig.tight_layout()
```

[35]:
```
plot_orbits(data)
```



The Earth is now on an eccentric orbit around the sound. By the spacing of the snapshots, which are constant in time, you can even tell that the Earth is faster around the perihelion and slower around the aphelion.

Adding Temperature

To calculate the temperature Earth and Sun need additional fields for their temperatures and their radii. In additional to that, we give the Earth an additional field for the Earth-Sun distance.

[36]:
```
R_Earth = c.R_earth.si.value
R_Sun   = c.R_sun.si.value
```

We initialize the Earth's temperature with zero and update it as soon as we have a function for it.

[37]:
```
sim.Earth.addfield("d", AU, description="Earth-Sun distance [m]")
sim.Earth.addfield("R", R_Earth, description="Radius [m]")
sim.Earth.addfield("T", 0., description="Temperature [K]")
```

[38]:
```
sim.Earth
```

[38]:
```
Group
-----
    d             : Field (Earth-Sun distance [m])
    M             : Field (Mass [kg]), constant
    r             : Field (Position [m])
    R             : Field (Radius [m])
    T             : Field (Temperature [K])
    v             : Field (Velocity [m/s])
  -----
```

[39]:
```
sim.Sun.addfield("R", R_Sun, description="Radius [m]", constant=True)
sim.Sun.addfield("T", 5778, description="Effective temperature [K]", constant=True)
```

[40]:
```
sim.Sun
```

[40]:
```
Group
-----
    M             : Field (Mass [kg]), constant
    r             : Field (Position [m])
    R             : Field (Radius [m]), constant
    T             : Field (Effective temperature [K]), constant
    v             : Field (Velocity [m/s])
  -----
```

3.1 Skipping Fields in Output Files

It is possible to not write certain fields into the output. This can be especially useful if you have large fields, that consume a significant amount of memory, that you don't need for your data analysis. Simply the the `save` attribute to `False`.

[41]:
```
sim.Sun.T.save = False
```

3.2 Updating fields

The energy the sun is emitting per time, i.e., it's luminosity L, is given by

$L_\odot = 4\pi R_\odot^2 \sigma_{\rm SB} T_{\rm eff}^4$.

The energy flux arriving at Earth, i.e., the Solar constant S is then given by

$S = \frac{L}{4\pi d^2}$.

Only one hemisphere of the Earth is illuminated at any time, so the energy that the Earth is receiving per unit time, neglecting any albedo effects, is given by

$P_{\rm in} = \pi R_\oplus^2 S$.

We assume that the temperature on Earth is in euilibrium, i.e., there is no difference between day and night. The energy the Earth is emitting per unit time is then given by

$P_{\rm out} = 4\pi R_\oplus^2 \sigma_{\rm SB} T^4$.

In equilibrium both are equal ($P_{\rm in} = P_{\rm out}$) and we can solve for T

$T = T_{\text{eff}} \sqrt[4]{\frac{R_\odot^2}{4d^2}}$.

So the equilibrium temperature on Earth depends on the Sun's effective temperature T_{eff}, the Sun's radius $R_\odot$, and the Earth-Sun distance d. We can now write a function that takes the frame object as argument and returns Earth's temperature.

[42]:
```
sigma_sb = c.sigma_sb.si.value
```

[43]:
```
def T(frame):
    return frame.Sun.T * (frame.Sun.R**2 / (4.*frame.Earth.d**2))**0.25
```

We can now assign this function to the updater of the temperature field

[44]:
```
sim.Earth.T.updater = T
```

The function for calculating the Earth-Sun distance is pretty simple and can be simply assigned to the updater of the field.

[45]:
```
def d(sim):
    return np.linalg.norm(sim.Earth.r - sim.Sun.r)
```

[46]:
```
sim.Earth.d.updater = d
```

Right now the temperature is zero as we initialized it.

[47]:
```
sim.Earth.T
```

[47]:
```
0.0
```

We can now use our the updater of the temperature field to calculate it's initial value

[48]:
```
sim.Earth.T.update()
```

[49]:
```
sim.Earth.T
```

[49]:
```
278.6190681198289
```

3.3 Updating groups

But before we can rerun the simulation we have to tell `simframe` how to update groups. This is not done automatically, because for one, not all fields and groups need to be updated, and second, the order of update matters. In our case, we do not need to update the group of the Sun, because it's temperature is not changing, and we need to update the Earth-Sun distance before we update the Earth's temperature, because we need the distance for it.

The only update operation that is performed once per time step is the update of the frame object `Frame.update()`. From here we have to delegate tasks down the tree structure.

There are in principle two methods of updating groups. One is by writing a function and assigning it to the group's updater. Here we write a function that is calling Earth's updater and assign it to theupdater of the frame.

[50]:
```
def update_Earth(frame):
    frame.Earth.update()
```

[51]:
```
sim.updater = update_Earth
```

The second method is by assigning a list of group attributes to an updater. The updater is then calling the updaters of these attributes in exactly that order.

[52]:
```
sim.Earth.updater = ["d", "T"]     # Earth-Sun distance first, then temperature
```

If the updater of a group has been set with a list, the order can be displayed. The attribute will return `None`, if no updater is set or if the updater was set with a function.

[53]:
```
sim.Earth.updateorder
```

[53]:
```
['d', 'T']
```

Resetting to initial conditions

We can now reset to the initial conditions we have saved earlier, reset the namespace writer, and run the simulation again.

[54]:
```
sim.Earth.r = r_Earth_ini
sim.Earth.v = v_Earth_ini
sim.Sun.r   = r_Sun_ini
sim.Sun.v   = v_Sun_ini
sim.t       = 0.
sim.writer.reset()
```

[55]:
```
sim.run()
```

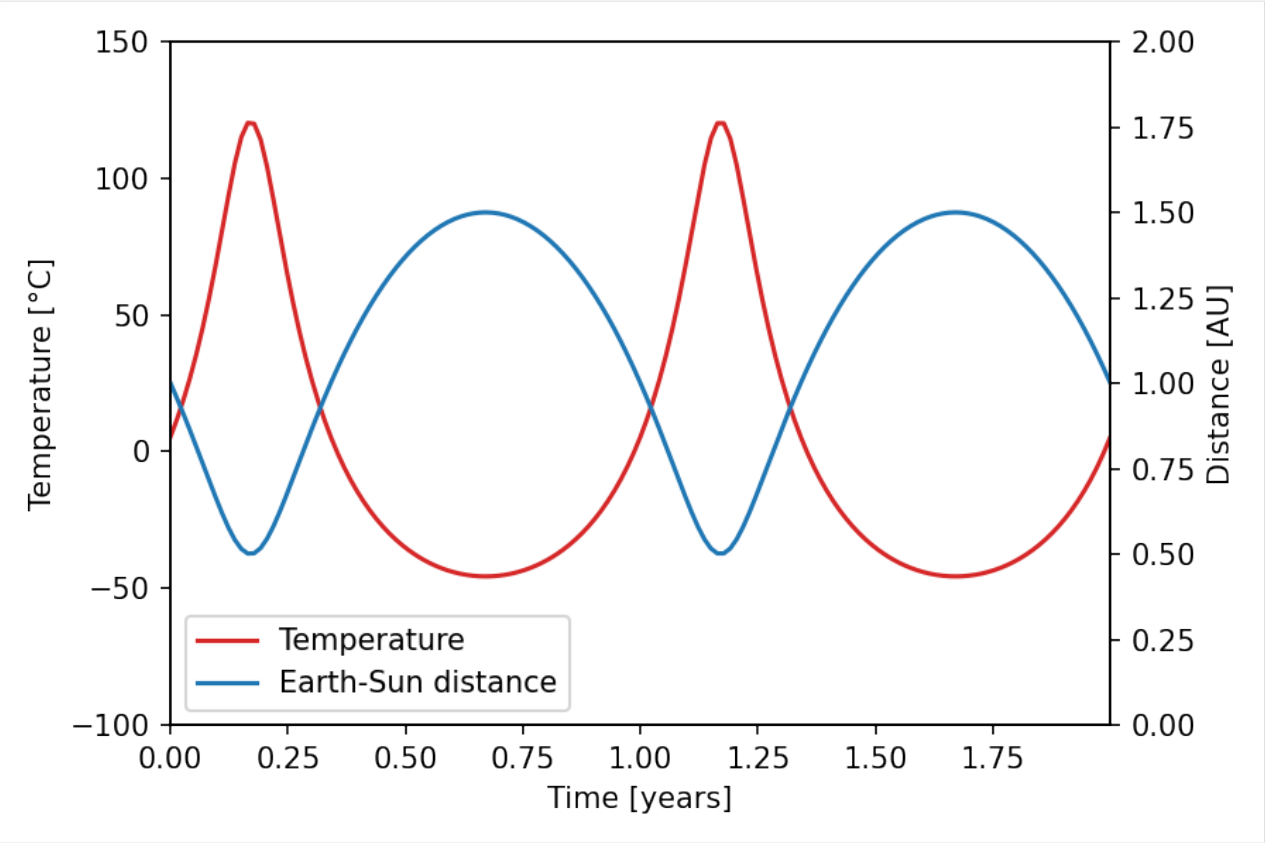
```
Execution time: 0:00:02
```

Reading and plotting

[56]:
```
data_T = sim.writer.read.all()
```

[57]:
```
def plot_temperature(data):
    fig, ax = plt.subplots(dpi=150)
    ax.plot(data.t/year, (data.Earth.T*u.K).to(u.Celsius, equivalencies=u.
↪temperature()), c="C3", label="Temperature")
    ax.plot(-2., 0., label="Earth-Sun distance", c="C0")
    ax.set_xlabel("Time [years]")
    ax.set_ylabel("Temperature [°C]")
    ax.set_xlim(data.t[0]/year, data.t[-1]/year)
    ax.set_ylim(-100., 150.)
    axr = ax.twinx()
    axr.plot(data.t/year, data.Earth.d/AU, c="C0")
    axr.set_ylabel("Distance [AU]")
    axr.set_ylim(0., 2.0)
    ax.legend()
    fig.tight_layout()
```

[58]:
```
plot_temperature(data_T)
```



3.4 The heartbeat concept

Every updater is actually performing three operations. The updater operation itself, a **"systole"**, that is executed before the update operation and a **"diastole"** that is executed after the update operation.

In this simple case we want to print Earth's temperature before and after it's calculation.

[59]:
```
def T_sys(frame):
    msg = "{:6s}: T = {:5.1f} K".format("Before", sim.Earth.T)
    print(msg)
```

[60]:
```
def T_dia(frame):
    msg = "{:6s}: T = {:5.1f} K".format("After", sim.Earth.T)
    print(msg)
```

We can assign these functions to systole and diastole of the temperature updater.

[61]:
```
sim.Earth.T.updater.systole = T_sys
sim.Earth.T.updater.diastole = T_dia
```

[62]:
```
sim.Earth.T = 0.
```

[63]:
```
sim.Earth.T.update()
```

```
Before: T =   0.0 K
After : T = 278.0 K
```

Note: The updater of an integration variable is not directly setting its new value. Only the integrator is setting the new value of the integration variable after all integration instructions have been executed successfully. This has the consequence, that the diastole of an integration variable *DOES NOT* have access to the updated value.

In this tutorial you'll learn how to

- create custom integration schemes

CHAPTER

FOUR

CUSTOM INTEGRATION SCHEMES

In this tutorial we want to estimate π with the following equation:

$\pi = 4 \int_0^1 \sqrt{1 - t^2} \mathrm{d}t$

We set up the simulation frame as explained in the previous tutorials.

[1]:
```
from simframe import Frame
```

[2]:
```
sim = Frame()
```

Adding field for :math:`pi` and integration variable

[3]:
```
sim.addfield("pi", 0., description="Approximation of pi")
sim.addintegrationvariable("t", 0.)
```

Differentiator

[4]:
```
import numpy as np

def f(frame, x, Y):
    return 4.*np.sqrt(1-x**2)
```

[5]:
```
sim.pi.differentiator = f
```

Step size

We set the step size to 0.25, i.e., the integral function is only evaluated four times in the simulation.

[6]:
```
def dt(frame):
    return 0.25
```

[7]:
```
sim.t.updater = dt
```

We do not need to write outputs for this model. We only have to tell the integrator when to stop the simulation, i.e., the upper bound of the integral.

[8]:
```
sim.t.snapshots = [1.]
```

Setting the integrator

[9]:
```
from simframe import Integrator
from simframe import Instruction
from simframe import schemes
```

[10]:
```
sim.integrator = Integrator(sim.t)
```

[11]:
```
sim.integrator.instructions = [Instruction(schemes.expl_1_euler, sim.pi)]
```

Running the simulation

[12]:
```
sim.run()
```

```
Execution time: 0:00:00
```

Results

[13]:
```
from IPython.display import Markdown as md
def print_table(sim):
    return md("| |$\pi$|rel. error|\n|-|-|-|\n|real|{:10.8f}||\n|approx.|{:10.8f}|{:9.
↪3e}|".format(np.pi,sim.pi,np.abs(np.pi-sim.pi)/np.pi))
```

[14]:
```
print_table(sim)
```

[14]:

	π	rel. error
real	3.14159265	
approx.	3.49570907	1.127e-01

The relative error is about 11 %.

There are in principle two way how we can improve this estimate. One is to decrease the stepsize. You can rerun the notebook with a smaller step size and compare the results.

Another method is to use more sophisticated integration schemes. We are now going to implement the 2nd-order Midpoint Method and the 4th-order Runge-Kutta method.

4.1 Writing custom integration schemes

All we have to do is write a function that takes the current value of the integration variable **x0**, the current value of the variable to be integrated **Y0**, and the step size **dx**. The function needs to return the change **dY** of our variable **Y0** after the integration. We can access the derivative of the variable with `Y0.derivative(x, Y)`.

For the midpoint method this looks as follows.

[15]:
```
def midpoint(x0, Y0, dx, *args, **kwargs):
    x1 = x0 + 0.5*dx
    Y1 = Y0 + 0.5*dx*Y0.derivative(x0, Y0)
    return dx*Y0.derivative(x1, Y1)
```

All we have to do now is to create an integration scheme from this function. This can be done by using `AbstractScheme` provided by `simframe`.

[16]:
```
from simframe.integration import Scheme
```

[17]:
```
expl_2_midpoint = Scheme(midpoint)
```

`expl_2_midpoint` is now our new integration scheme. The naming convention is `<expl/impl>_<order>_<name><_other>`.

We can now assign a new instruction set using our new method to the integrator just as with the 1st-order Euler method.

[18]:
```
sim.integrator.instructions = [Instruction(expl_2_midpoint, sim.pi)]
```

Before we restart the simulation we have to reset to the initial conditions.

[19]:
```
sim.t = 0
sim.pi = 0
```

[20]:
```
sim.run()
```

```
Execution time: 0:00:00
```

[21]:
```
print_table(sim)
```

[21]:

	π	rel. error
real	3.14159265	
approx.	3.18392922	1.348e-02

The error is now reduced to 1.3 % only by using a higher order method.

Note: the higher order method needs more operations and is therefore slower. This does not really matter in this case, but might be important for more complex simulations.

4th-order Runge-Kutta

The scheme function for the 4th-order explicit Runge-Kutta method looks as follows.

[22]:
```
def rk4(x0, Y0, dx, *args, **kwargs):
    k1 = Y0.derivative(x0          , Y0            )
    k2 = Y0.derivative(x0 + 0.5*dx, Y0 + 0.5*dx*k1)
    k3 = Y0.derivative(x0 + 0.5*dx, Y0 + 0.5*dx*k2)
    k4 = Y0.derivative(x0 +     dx, Y0 +     dx*k3)
    return dx*(1/6*k1 + 1/3*k2 + 1/3*k3 + 1/6*k4)
```

[23]:
```
expl_4_rungekutta = Scheme(rk4)
```

[24]:
```
sim.integrator.instructions = [Instruction(rk4, sim.pi)]
```

[25]:
```
sim.t = 0
sim.pi = 0
```

[26]:
```
sim.run()
```

```
Execution time: 0:00:00
```

[27]:
```
print_table(sim)
```

[27]:

	π	rel. error
real	3.14159265	
approx.	3.12118917	6.495e-03

With this method the error is reduced down to 0.6 %.

4.2 Available integration schemes

But before you take effort into developing your own integration schemes, take a look at the schemes already provided by `simframe`:

[28]:
```
_  = "|Scheme|Description|\n"
_ += "|------|-----------|\n"
for s in schemes.__dir__():
    if s == "update": continue
    if not s.startswith("_"): _ += "|"+s+"|"+schemes.__dict__[s].description+"|\n"
md(_)
```

[28]:

Scheme	Description
expl_1_euler	Explicit 1st-order Euler method
expl_2_fehlberg_adptv	Explicit adaptive 2nd-order Fehlberg's method
expl_2_heun	Explicit 2nd-order Heun's method
expl_2_heun_euler_adptv	Explicit adaptive 2nd-order Heun-Euler method
expl_2_midpoint	Explicit 2nd-order midpoint method
expl_2_ralston	Explicit 2nd-order Ralston's method
expl_3_bogacki_shampine_adptv	Explicit adaptive 3rd-order Bogacki-Shampine method
expl_3_gottlieb_shu_adptv	Explicit adaptive 3rd-order Gottlieb-Shu method
expl_3_heun	Explicit 3rd-order Heun's method
expl_3_kutta	Explicit 3rd-order Kutta's method
expl_3_ralston	Explicit 3rd-order Ralston's method
expl_3_ssprk	Explicit 3rd-order Strong Stability Preserving Runge-Kutta method
expl_4_38rule	Explicit 4th-order 3/8 rule method
expl_4_ralston	Explicit 4th-order Ralston's method
expl_4_runge_kutta	Explicit 4th-order classical Runge-Kutta method
expl_5_cash_karp_adptv	Explicit adaptive 5th-order Cash-Karp method
impl_1_euler_direct	Implicit 1st-order direct Euler method
impl_1_euler_gmres	Implicit 1st-order Euler method with GMRES solver
impl_2_midpoint_direct	Implicit 2nd-order direct midpoint method

In this tutorial you'll learn how to

- set up adaptive integration schemes
- set up fail operations
- set preparation and finalization instructions to the integrator
- use suggested step sizes

For this notebook you need `matplotlib` in addition to the `simframe` requirements.

CHAPTER

FIVE

ADAPTIVE INTEGRATION SCHEMES

For this tutorial we revisit the problem of the first tutorial.

- $\frac{\mathrm{d}Y}{\mathrm{d}x} = b\,Y$
- $Y\,(0) = A$
- $Y\,(x) = A\,e^{bx}$

But this time we increase the step size, such that the numeric solution is oscillating.

Problem parameters

[1]:
```
dx =  1.75
A  =  1.
b  = -1.
```

Setting up frame

[2]:
```
from simframe import Frame

sim = Frame(description="Adaptive step sizing")
```

Adding field and integration variable

[3]:
```
sim.addintegrationvariable("x", 0.)
sim.addfield("Y", A)
```

Setting up writer

[4]:
```
from simframe import writers
```

[5]:
```
sim.writer = writers.namespacewriter
sim.writer.dumping = False
sim.writer.verbosity = 0
```

Setting differential equation

We slightly modify the differential equation and add a counter that tells us how often the function got called.

[6]:
```
N = 0
```

[7]:
```
def dYdx(frame, x, Y):
    global N
    N += 1
    return b*Y
```

[8]:
```
sim.Y.differentiator = dYdx
```

Setting up the step size

In this example we return a constant step size defined previously.

[9]:
```
def fdx(frame):
    return dx
```

[10]:
```
sim.x.updater = fdx
```

Setting up snapshots

[11]:
```
import numpy as np
sim.x.snapshots = np.arange(dx, 15., dx)
```

Setting up integrator

First we simply integrate with the known 1st-order Euler scheme with a constant step size.

[12]:
```
from simframe import Integrator
from simframe import Instruction
from simframe import schemes
```

[13]:
```
sim.integrator = Integrator(sim.x)
```

[14]:
```
sim.integrator.instructions = [Instruction(schemes.expl_1_euler, sim.Y)]
```

Running the simulation

[15]:
```
sim.run()
```

```
Execution time: 0:00:00
```

Reading data

[16]:
```
data = sim.writer.read.all()
```

We store the data in a list for later comparison.

[17]:
```
dataset = []
dataset.append([data, "Euler 1st-order", N])
```

Plotting

Function returning the exact solution used for plotting

[18]:
```
def f(x):
    return A*np.exp(b*x)
```

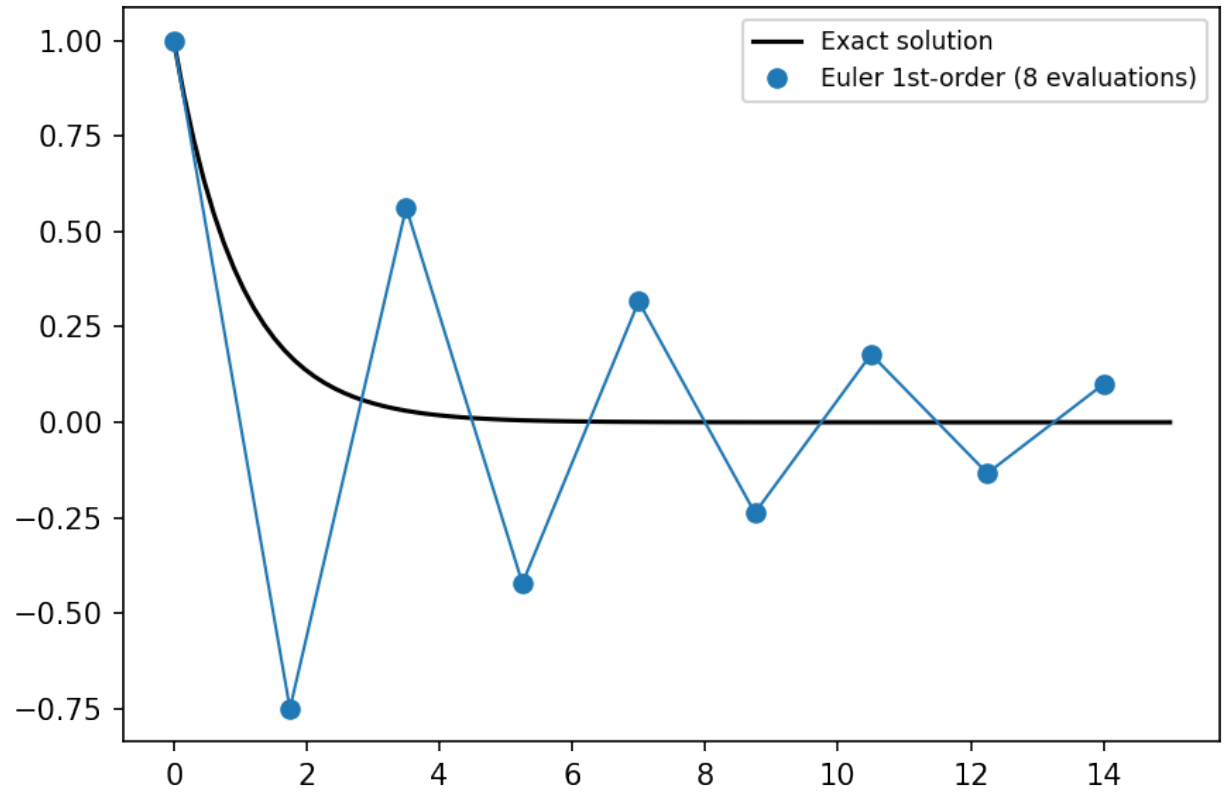
[19]:
```
import matplotlib.pyplot as plt

def plot(dataset):
    fig, ax = plt.subplots(dpi=150)
    x = np.linspace(0, 15., 100)
    ax.plot(x, f(x), c="black", label="Exact solution")
    for i, val in enumerate(dataset):
```

(continues on next page)

(continued from previous page)

```
        ax.plot(val[0].x, val[0].Y, "o", c="C"+str(i), label="{} ({} evaluations)".
↪format(val[1], val[2]))
        ax.plot(val[0].x, val[0].Y, c="C"+str(i), lw=1)
    ax.legend(fontsize="small")
    fig.tight_layout()
```

[20]:
```
plot(dataset)
```



As you can see, the calculated solution is oscillating aroung the real solution.

5.1 Adaptive Step Sizing Schemes

Instead of a constant step size, we now want to adjust it dynamically. If the error is too large, we decrease the step size. As an estimate for the error we compare the full Euler 1st-order step to the solution we get from performing two consecutive Euler 1st-order step with half the step size. If the relative error is larger than 10 %, we repeat the integration with a smaller step size until we are withing the error.

We therefore have to set up a custom integration scheme as was shown in the previous example. The scheme has to perform the full step and two semi steps and and needs to compare them. If the error is too large, the scheme has to return `False`. If it was successful, it has to return the change `dY` of the dependent variable `Y`.

[21]:
```
def adaptive(x0, Y0, dx, *args, **kwargs):
    fullstep  = dx*Y0.derivative(x0, Y0)
    semistep1 = 0.5*fullstep
    semistep2 = 0.5*dx*Y0.derivative(x0+0.5*dx, Y0+semistep1)
```

(continues on next page)

(continued from previous page)

```
    semisteps = semistep1 + semistep2

    relerr = np.abs((semisteps-fullstep)/(Y0+semisteps))

    if relerr > 0.1:
        return False
    else:
        return semisteps
```

Creating scheme and modifying instruction set

[22]:
```
from simframe.integration import Scheme
adaptive = Scheme(adaptive)
sim.integrator.instructions = [Instruction(adaptive, sim.Y)]
```

5.2 The fail operation

If the integration failed, because the step size was too large, i.e., the scheme returned `False`, the integrator triggers a fail operation. This operation can be used to manipulate the step size. In our case, we want to decrease the step size by a factor of 10.

Note the `global dx` to manipule `dx` persistently outside the function.

[23]:
```
def failop(frame):
    global dx
    dx /= 10.
```

We assign this function to the fail operation of the integrator. The fail operation needs the parent `Frame` object as positional argument. If any instruction returns `False`, the fail operation will be executed and the integrator goes through the instructions again, before updating the fields.

Note: In case you have an `update` instuction in your instruction set, you have to undo it by yourself in the fail operation.

[24]:
```
sim.integrator.failop = failop
```

5.3 Preparation and finalization

If the integration was successful, we want to increase our step size by a factor of 5. This is done via the `finalizer` of the integrator, which is called after going through the instruction set and after updating the fields to be integrated. The equivalent that is called before going through the instructions set is `<Frame>.integrator.preparator`.

[25]:
```
def finalize(frame):
    global dx
    dx *= 5.
```

[26]:
```
sim.integrator.finalizer = finalize
```

Resetting the parameters

[27]:
```
N = 0
sim.x = 0.
sim.Y = 1.
sim.writer.reset()
```

Before running the simulation we save the inital step size for later use.

[28]:
```
import copy
dx_ini = dx
```

Running the simulation

[29]:
```
sim.run()
```

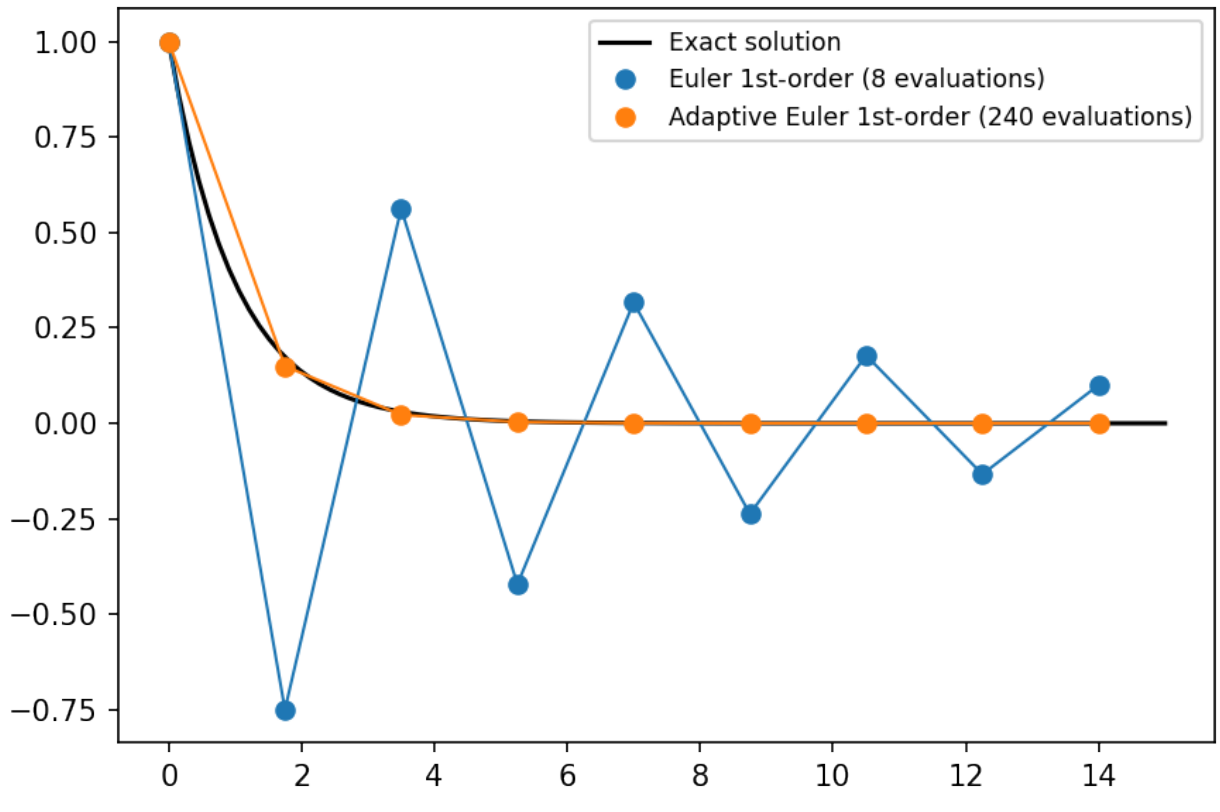
```
Execution time: 0:00:00
```

Reading data and plotting

[30]:
```
data_adaptive = sim.writer.read.all()
```

[31]:
```
dataset.append([data_adaptive, "Adaptive Euler 1st-order", N])
```

[32]:
```
plot(dataset)
```



Embedded methods

Another technique for estimating the error is to perform a higher order method for the full step instead of the same method for two semistep as done before. In some cases the higher order method is utilizing the result of the lower order method, saving evaluations. These methods are called embedded methods.

`simframe` comes with a few embedded methods. In this example we use the embedded Bogacki-Shampine method.

[33]:
```
sim.integrator.instructions = [Instruction(schemes.expl_3_bogacki_shampine_adptv, sim.
↪Y)]
```

5.4 Suggested step sizes

The embedded methods included in `simframe` provide an estimate for the new step size depending on the truncation error. This estimate is saved in the integration variable in `<IntVar>.suggested`. We have to modify the step size function to utilize this estimate.

[34]:
```
def fdx(sim):
    return sim.x.suggested
```

[35]:
```
sim.x.updater = fdx
```

And we have to give an initial suggestion for the step size. We'll use the initial value as before.

[36]:
```
sim.x.suggest(dx_ini)
```

Unsetting fail operation and finalization

The fail operation and finalization operations are not needed anymore and have to be unset.

[37]:
```
sim.integrator.failop = None
sim.integrator.finalizer = None
```

Resetting the parameters and running the simulation

[38]:
```
N = 0
sim.x = 0.
sim.Y = 1.
sim.writer.reset()
```

[39]:
```
sim.run()
```

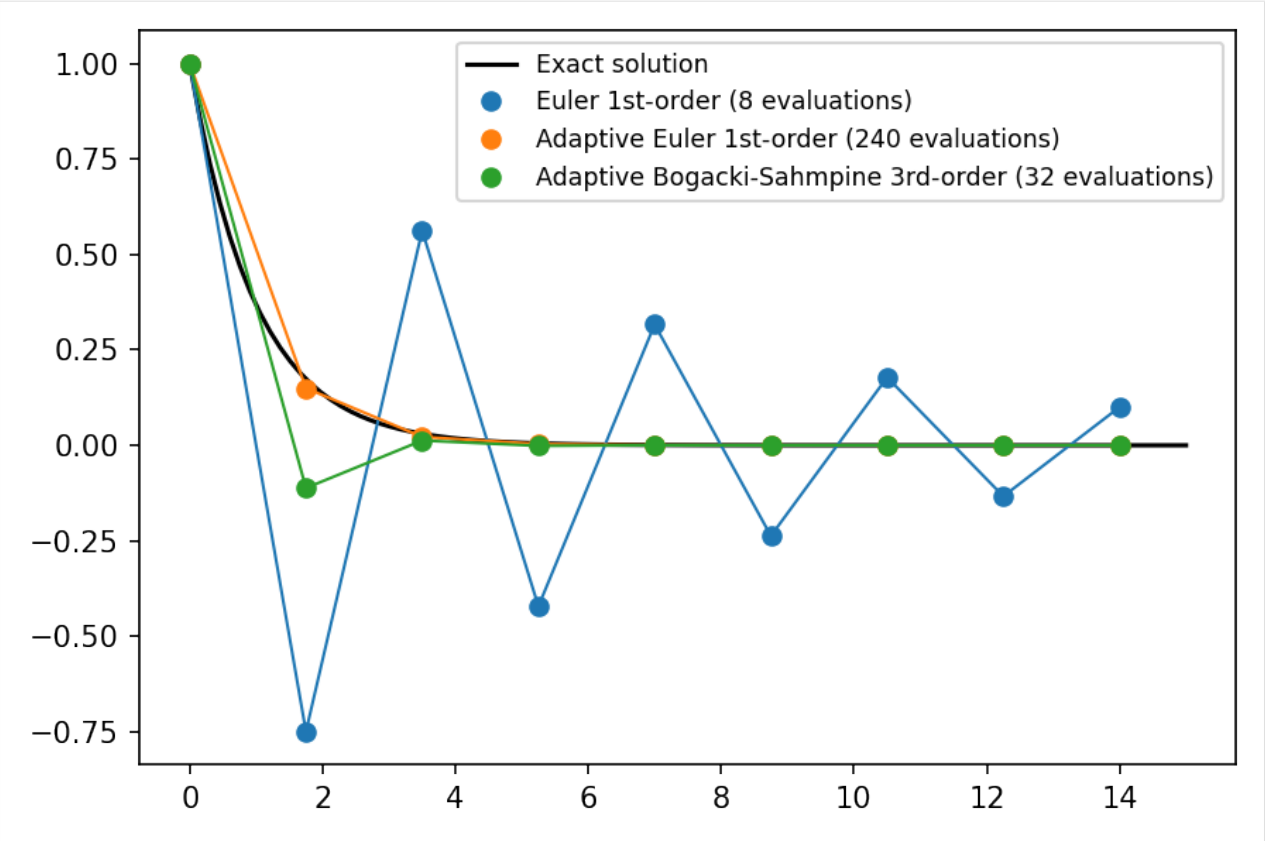
```
Execution time: 0:00:00
```

Reading data and plotting

[40]:
```
data_bogackishampine = sim.writer.read.all()
```

[41]:
```
dataset.append([data_bogackishampine, "Adaptive Bogacki-Sahmpine 3rd-order", N])
```

[42]:
```
plot(dataset)
```



As you can see the Bogacki-Shampine method needs 32 eveluations of the derivative in this setup. That's only about half of the amount of the adaptive Euler method and it's significantly more accurate.

Change the stepsize to `dx = 2.25` and run the notebook again. For this step size the Euler 1st-order method is unstable.

5.5 Passing keyword arguments to integration scheme

It is also possible to pass key word arguments to the integrator to control its behaviour. We could for example increase the accuracy by changing the desired relative error. This is done by passing a controller dictionary to the instruction.

[43]:
```
sim.integrator.instructions = [Instruction(schemes.expl_3_bogacki_shampine_adptv, sim.
↪Y, controller={"eps": 0.01})]
```

Here we want to have maximum relative error of $1\,\%$. Default is $10\,\%$.

Resetting the parameters and running the simulation

Note the `reset=True` keyword when resetting the suggested step size. Suggesting step sizes takes the minimum of the suggested and the current value, which is still set from the earlier integration. We therefore have to reset the current value with the keyword.

[44]:
```
N = 0
sim.x = 0.
sim.x.suggest(dx_ini, reset=True)
sim.Y = 1.
```

(continues on next page)

(continued from previous page)

```
sim.Y._buffer = None
sim.writer.reset()
```

[45]:
```
sim.run()
```

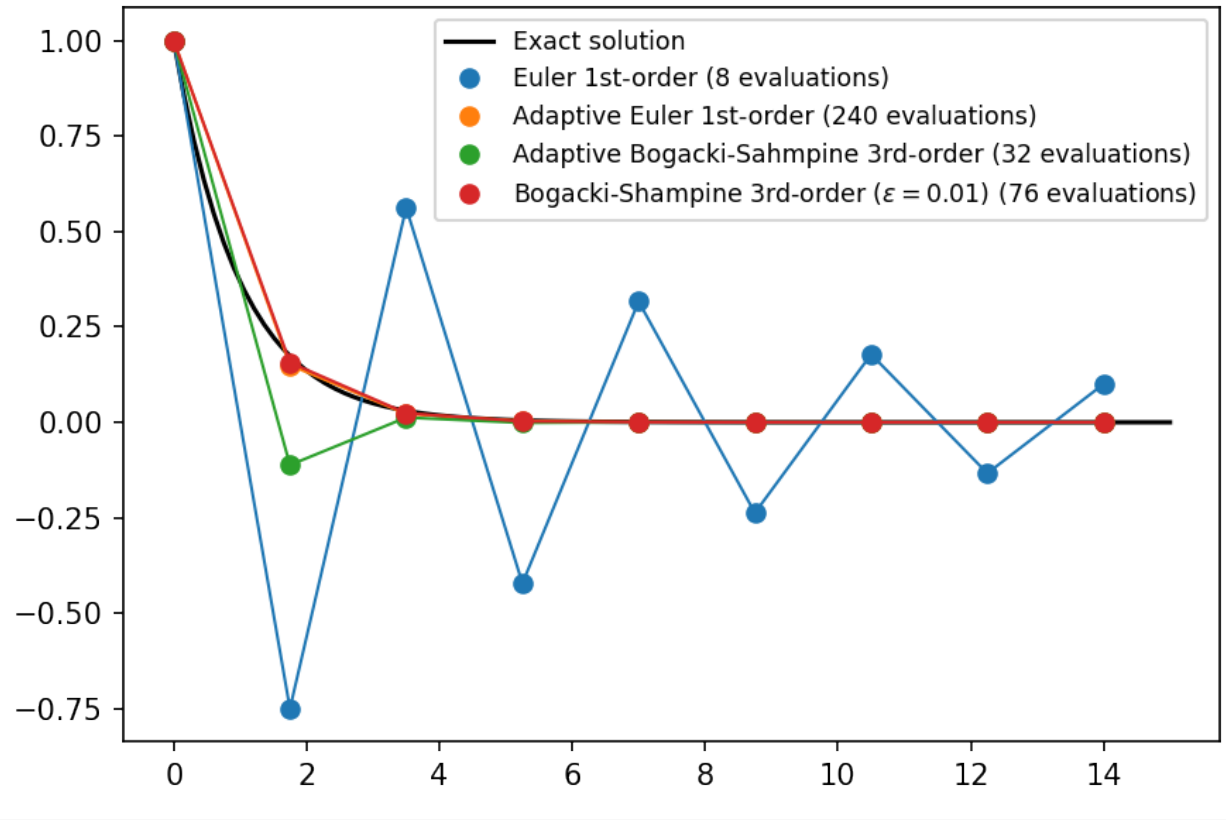
```
Execution time: 0:00:00
```

Reading and plotting data

[46]:
```
data_bogackishampine_accurate = sim.writer.read.all()
```

[47]:
```
dataset.append([data_bogackishampine_accurate, "Bogacki-Shampine 3rd-order ($\
↪epsilon=0.01$)", N])
```

[48]:
```
plot(dataset)
```



With this settings it takes slightly longer to approach the analytical solution, but the overshooting in the beginning is prevented.

In this tutorial you'll learn how to

- set up implicit integration schemes

For this notebook you need `matplotlib` in addition to the `simframe` requirements.

CHAPTER

SIX

IMPLICIT INTEGRATION

In this example we revisit the differential equation from the first tutorial and the last tutorial about adaptive scheme.

For this tutorial we revisit the problem of the first tutorial.

- $\frac{\mathrm{d}Y}{\mathrm{d}x} = b\,Y$
- $Y\,(0) = A$
- $Y\,(x) = A\,e^{bx}$

But this time we increase the step size, such that the 1st-order Euler scheme is unstable.

Analytic solution

[1]:
```
import numpy as np
```

[2]:
```
def f(x, A, b):
    return np.exp(b*x)
```

Model parameters

[3]:
```
A  =   1.
b  =  -1.
dx =  10.
```

This time we chose a large step size and set up the frame.

[4]:
```
from simframe import Frame
```

[5]:
```
sim_expl = Frame(description="Explicit integratioon")
```

[6]:
```
sim_expl.addfield("Y", A)
sim_expl.addintegrationvariable("x", 0.)
```

[7]:
```
def fdx(frame):
    return dx
sim_expl.x.updater = fdx
sim_expl.x.snapshots = [10.]
```

[8]:
```
def diff_expl(frame, x, Y):
    return b*Y
sim_expl.Y.differentiator = diff_expl
```

[9]:
```
from simframe import Integrator
from simframe import Instruction
from simframe import schemes
```

[10]:
```
sim_expl.integrator = Integrator(sim_expl.x)
sim_expl.integrator.instructions = [Instruction(schemes.expl_1_euler, sim_expl.Y)]
```

[11]:
```
from simframe import writers
```

[12]:
```
sim_expl.writer = writers.namespacewriter
sim_expl.writer.dumping = False
sim_expl.writer.verbosity = 0
```

[13]:
```
sim_expl.run()
```

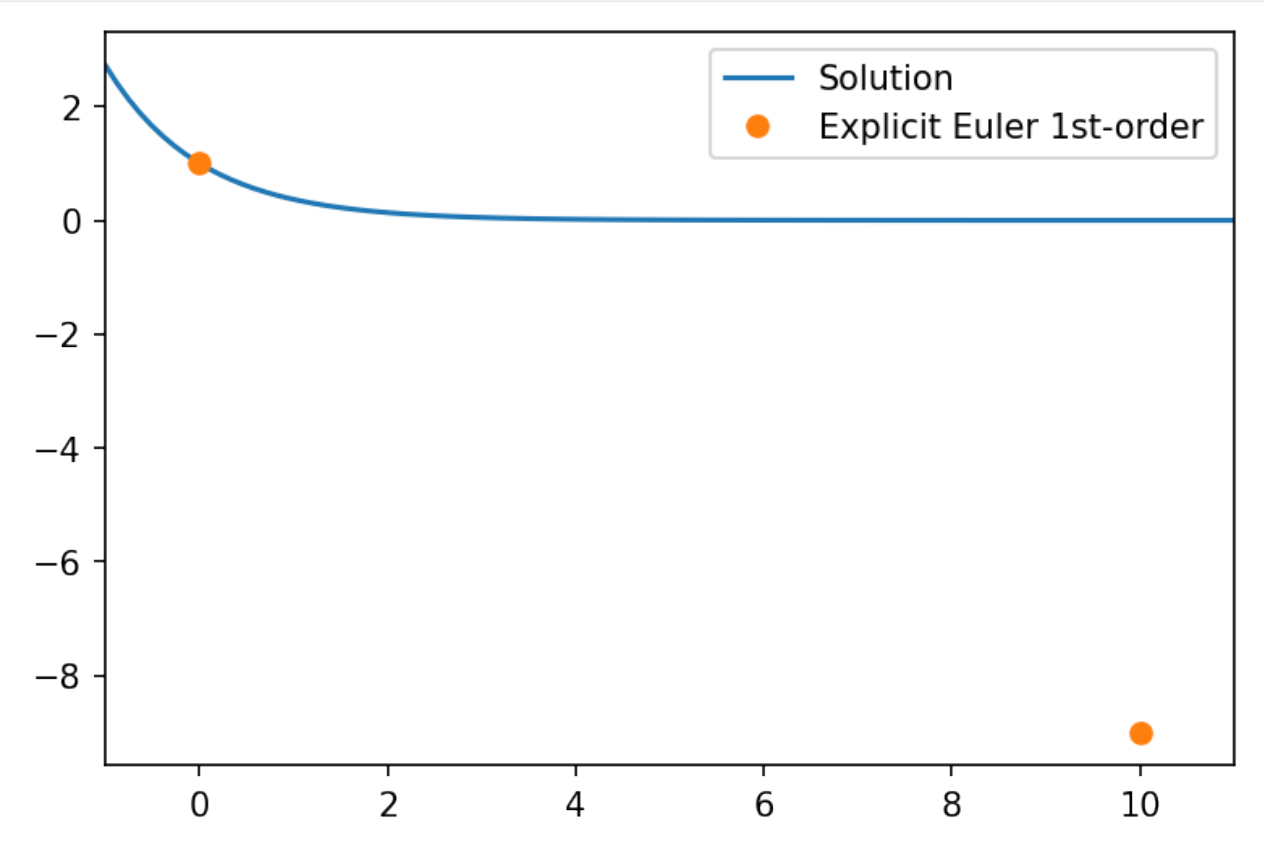
```
Execution time: 0:00:00
```

Reading data and plotting

[14]:
```
data_expl = sim_expl.writer.read.all()
```

[15]:
```
import matplotlib.pyplot as plt

def plot(ls):
    fig, ax = plt.subplots(dpi=150)
    x = np.linspace(-1., 11, 100)
    ax.set_xlim(x[0], x[-1])
    ax.plot(x, f(x, A, b), label="Solution")
    for sim, d in ls:
        ax.plot(sim.x, sim.Y, "o", label=d)
    ax.legend()
    plt.show()
```

[16]:
```
plot([(data_expl, "Explicit Euler 1st-order")])
```



In this case the step width is way too large to produce a useful result.

The only help would be to reduce the step size or to go for implicit integration.

6.1 Background: Implicit integration

For explicit integration the derivative f of the differential equation is evaluated at the current point in time or space:

$\frac{\Delta Y}{\Delta x} = \frac{Y_{n+1} - Y_n}{\Delta x} = f\left(Y_n\right),$

which leads to the simple 1st-order Euler scheme.

$Y_{n+1} = Y_n + \Delta x \, f\left(Y_n\right).$

Implicit integration means that the derivative is evaluated at the future point in time or space:

$\frac{Y_{n+1} - Y_n}{\Delta x} = f\left(Y_{n+1}\right).$

What looks ridiculous at first is mathematically sound.

Imagine that the derivative can be written as a matrix equation.

$f\left(\vec{Y}\right) = \mathbf{J} \cdot \vec{Y},$

with the Jacobian matrix $\mathbf{J}$.

Plugging this into our differential equation yields

$\vec{Y}_{n+1} - \vec{Y}_n = \Delta x \, \mathbf{J} \cdot \vec{Y}_{n+1}$

$\Leftrightarrow (\mathbb{1} - \Delta x\, \mathbb{J}) \cdot \vec{Y}_{n+1} = \vec{Y}_n$

$\Leftrightarrow \vec{Y}_{n+1} = (\mathbb{1} - \Delta x\, \mathbb{J})^{-1} \cdot \vec{Y}_n$

The solution can be found by inverting the matrix $\mathbb{1} - \Delta x\, \mathbb{J}$.

In our simple case this translates to

$\mathbb{J} = \begin{pmatrix} b \end{pmatrix}$

and

$Y_{n+1} = \frac{1}{1-\Delta x\, b} Y_n$

For large step sizes ($\Delta\, x \to \infty$) this goes to zero ($Y_n \to 0$) as it should compared to the exact solution. The integration scheme is *"unconditionally stable"*.

6.2 Setting up implicit integration

Setting up implicit integration is similar to explicit integration. We therefore just copy our frame and reset the values.

[17]:
```
import copy
```

[18]:
```
sim_impl = copy.deepcopy(sim_expl)
sim_impl.x = 0
sim_impl.Y = A
sim_impl.writer.reset()
```

The important difference is now, that instead of the derivative we have to provide the Jacobian $\mathbb{J}$ to our field Y, which is in our case very simple. The function for the Jacobi matrix needs the parent frame object as first and the integration variable as second positional argument.

[19]:
```
def jac_impl(sim, x):
    return np.array([b])
sim_impl.Y.jacobinator = jac_impl
```

We can now use an implicit scheme in our instruction set.

[20]:
```
sim_impl.integrator.instructions = [Instruction(schemes.impl_1_euler_direct, sim_impl.
↪Y)]
```

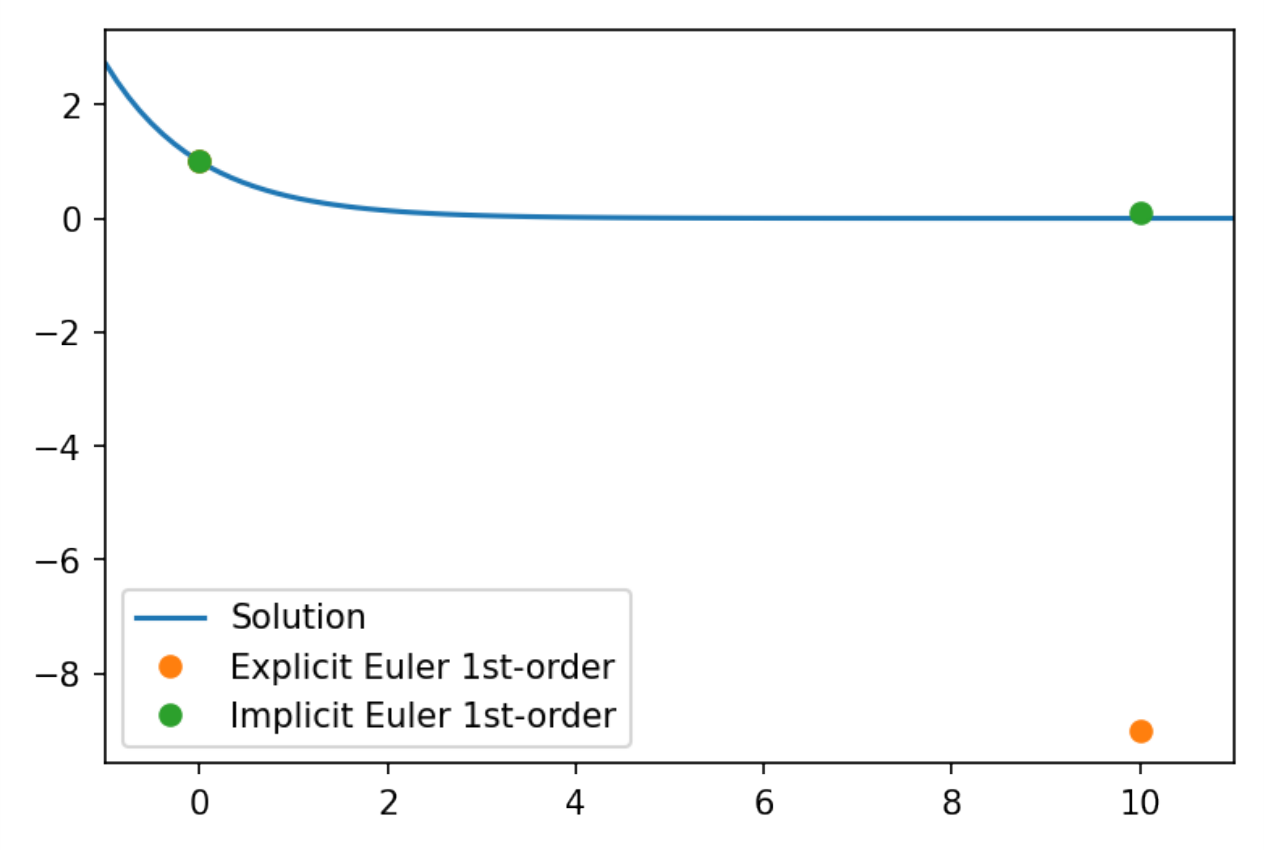
Now we can rerun the simulation.

[21]:
```
sim_impl.run()
```

```
Execution time: 0:00:00
```

[22]:
```
data_impl = sim_impl.writer.read.all()
```

[23]:
```
plot([(data_expl, "Explicit Euler 1st-order"),(data_impl, "Implicit Euler 1st-order
↪")])
```



Since implicit schemes involve matrix inversions, it can be very costly. The method shown here uses `numpy.linalg.inv()` to compute the inverse matrix, which is basically Gaussian elimination with LU factorization. There are other methods that might be more suitable for your problem.

Note: If the jacobinator is set, but not the differentiator, `simframe` will try to calculated the derivative from the Jacobi matrix by assuming

$\vec{Y}' = \mathbb{J} \cdot \vec{Y}$

[24]:
```
sim_impl.Y.differentiator = None     # unsetting the differentiator
```

[25]:
```
sim_impl.Y.derivative()
```

[25]:
```
-0.09090909090909094
```

Only if neither the differentiator, nor the jacobinator are set, `Field.derivative()` will return zeros in the shape of the field, i.e., the derivative is zero.

[26]:
```
sim_impl.Y.jacobinator = None
```

[27]:
```
sim_impl.Y.derivative()
```

[27]:
```
0.0
```

This tutorial is a showcase for the potential of `simframe`.

For this notebook you need `matplotlib` in addition to the `simframe` requirements.

CHAPTER

SEVEN

COUPLED OSCILLATORS

In this example we'll have a look at coupled oscillations of two bodies with masses m_i connected via three springs with themselves and attached to walls. The goal is to calculate the time evolution of this system if it's not in equilibrium. The springs have spring constants of k_i and lengths of l_i when no forces are acting on them. The distance between the walls is L.

[1]:
```
k1, l1 = 1., 6.
k2, l2 = 2., 6.
k3, l3 = 1., 6.

m1, m2 = 1., 1.

L = 15.
```

Spring connected serially have a resulting spring constant of K, which is the inverse sum of the individual spring constants.

[2]:
```
Kinv = 1./k1 + 1./k2 + 1./k3
K    = 1./Kinv
```

The force exerted by a spring is given by

$\vec{F} = -k \cdot \vec{d}$

where $\vec{d}$ is the displacement vecor from its equilibrium position. The system can be more easily solved by solving for the time evolution of the displacements $\vec{d_i}$ of the bodies. To convert it back into actual coordinates, we first have to finde the equilibrium positions $\vec{x_i}$ of the bodies.

If the system is in equilibrium, the forces acting on each individual spring are identical.

[3]:
```
F = - ( L - ( l1 + l2 + l3 ) ) * K
```

From this we can calculate the equilibrium positions.

[4]:
```
x1 = l1 - F/k1
```

[5]:
```
x2 = F/k3 + L - l3
```

In equilibrium our system looks as follows.

[6]:
```
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import numpy as np
```

[7]:
```
def getspring(l1, l2, bw):
    L = l2 - l1
    d = L/6.
    x = np.array([l1, d, d/2., d, d, d, d/2., d], dtype=np.float)
    for i in range(1, 8):
        x[i] += x[i-1]
    y = np.array([0., 0., 2.*bw, -2.*bw, 2.*bw, -2*bw, 0., 0.], dtype=np.float)
    return x, y
```

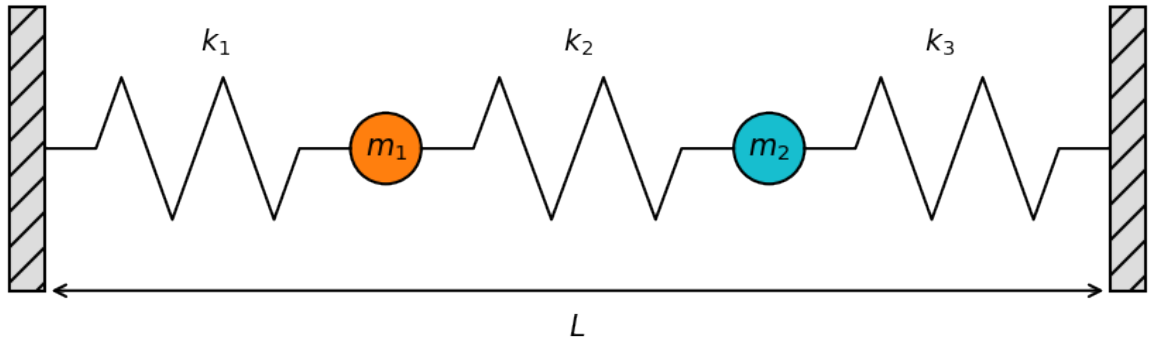
[8]:
```
def plot_system(bw, x1, x2, L):

    fig, ax = plt.subplots(dpi=150)
    ax.axis("off")
    ax.set_aspect(1.)

    rectl = patches.Rectangle((-bw, -4.*bw), bw, 8.*bw, linewidth=1, edgecolor="
↪#000000", facecolor="#dddddd", hatch="//")
    ax.add_patch(rectl)
    rectr = patches.Rectangle((L, -4.*bw), bw, 8.*bw, linewidth=1, edgecolor="#000000
↪", facecolor="#dddddd", hatch="//")
    ax.add_patch(rectr)
    body1 = patches.Circle((x1, 0), bw, linewidth=1, edgecolor="#000000", facecolor=
↪"C1")
    ax.add_patch(body1)
    body2 = patches.Circle((x2, 0), bw, linewidth=1, edgecolor="#000000", facecolor=
↪"C9")
    ax.add_patch(body2)

    s1x, s1y = getspring(0., x1-bw, bw)
    ax.plot(s1x, s1y, c="#000000", lw=1)

    s2x, s2y = getspring(x1+bw, x2-bw, bw)
    ax.plot(s2x, s2y, c="#000000", lw=1)

    s3x, s3y = getspring(x2+bw, L, bw)
    ax.plot(s3x, s3y, c="#000000", lw=1)

    ax.set_xlim(-2.*bw, L+2.*bw)
    ax.set_ylim(-3., 3.)

    fig.tight_layout()

    return fig, ax
```

[9]:
```
bw = 0.5
fig, ax = plot_system(bw, x1, x2, L)

ax.text((x1+0.)/2., 3*bw, "$k_1$", verticalalignment="center", horizontalalignment=
↪"center")
ax.text((x2+x1)/2., 3*bw, "$k_2$", verticalalignment="center", horizontalalignment=
↪"center")
ax.text((L +x2)/2., 3*bw, "$k_3$", verticalalignment="center", horizontalalignment=
↪"center")

ax.text(x1, 0., "$m_1$", verticalalignment="center", horizontalalignment="center")
ax.text(x2, 0., "$m_2$", verticalalignment="center", horizontalalignment="center")
```

(continues on next page)

(continued from previous page)

```
ax.annotate(text='', xy=(0., -4.*bw), xytext=(L, -4.*bw), arrowprops=dict(arrowstyle='
↪<->', lw=1))
ax.text(L/2, -5.*bw, "$L$", verticalalignment="center", horizontalalignment="center")

plt.show()
```



We know want to displace mass m_1 from its equilibrium position and calculate the time evolution of the whole system.

The force acting on m_1 is given by

$F_1 = m\dot{v_1} = -(k_1 + k_2) \cdot d_1 + k_2 \cdot d_2$

Vector notation is ommited since the problem is one-dimensional. The change in the displacement d_1 is given by

$\dot{d_1} = v_1$

Similarily for the second body

$F_2 = m\dot{v_2} = -(k_2 + k_3) \cdot d_2 + k_2 \cdot d_1$

$\dot{d_2} = v_2$

This is a system of coupled differential equations that can be written in matrix form

$$\begin{pmatrix} \dot{d_1} \\ \dot{d_2} \\ \dot{v_1} \\ \dot{v_2} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{k_1+k_2}{m_1} & \frac{k_2}{m_1} & 0 & 0 \\ \frac{k_2}{m_2} & -\frac{k_2+k_3}{m_2} & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} d_1 \\ d_2 \\ v_1 \\ v_2 \end{pmatrix}$$

or short

$\frac{\mathrm{d}}{\mathrm{d}t}\vec{Y} = \mathbf{J} \cdot \vec{Y}$

with the state vector $\vec{Y}$ and the Jacobian $\mathbf{J}$.

We can now start setting up the frame.

[10]:
```
from simframe import Frame
```

[11]:
```
sim = Frame(description="Coupled Springs")
```

[12]:
```
import numpy as np
```

[13]:
```
Y = np.array([-3., 0., 0., 0.])
sim.addfield("Y", Y, description="State vector")
```

In this configuration mass m_1 is displaced by 3 to the left while m_2 is in its equilibrium position. Both bodies are at rest.

We set up the time as integration variable.

[14]:
```
sim.addintegrationvariable("t", 0., description="Time")
```

[15]:
```
def dt(sim):
    return 0.1
```

[16]:
```
sim.t.updater = dt
```

[17]:
```
sim.t.snapshots = np.arange(.1, 50., .1)
```

In principle this would be enough to run the simulation. But for convenience we set up a few more fields and groups.

[18]:
```
sim.addgroup("b1", description="Body 1")
sim.addgroup("b2", description="Body 2")

# Body 1
sim.b1.addfield("m" , m1, description="Mass", constant=True)
sim.b1.addfield("d" , 0., description="Displacement")
sim.b1.addfield("x" , 0., description="Position")
sim.b1.addfield("x0", x1, description="Equilibrium Position", constant=True)
sim.b1.addfield("v" , 0., description="Velocity")

# Body 2
sim.b2.addfield("m" , m2, description="Mass", constant=True)
sim.b2.addfield("d" , 0., description="Displacement")
sim.b2.addfield("x" , 0., description="Position")
sim.b2.addfield("x0", x2, description="Equilibrium Position", constant=True)
sim.b2.addfield("v" , 0., description="Velocity")
```

These fields need to be updated from the state vector.

[19]:
```
# Body 1
def update_d1(sim):
    return sim.Y[0]
sim.b1.d.updater = update_d1
def update_v1(sim):
    return  sim.Y[2]
sim.b1.v.updater = update_v1
def update_x1(sim):
    return sim.b1.x0 + sim.b1.d
sim.b1.x.updater = update_x1

# Body 2
def update_d2(sim):
    return sim.Y[1]
sim.b2.d.updater = update_d2
def update_v2(sim):
    return  sim.Y[3]
sim.b2.v.updater = update_v2
```

(continues on next page)

(continued from previous page)

```
def update_x2(sim):
    return sim.b2.x0 + sim.b2.d
sim.b2.x.updater = update_x2
```

And we are adding more groups for the spring parameters.

[20]:
```
sim.addgroup("s1", description="Spring 1")
sim.addgroup("s2", description="Spring 2")
sim.addgroup("s3", description="Spring 3")

sim.s1.addfield("k", k1, description="Spring Constant", constant=True)
sim.s1.addfield("l", l1, description="Length", constant=True)
sim.s2.addfield("k", k2, description="Spring Constant", constant=True)
sim.s2.addfield("l", l2, description="Length", constant=True)
sim.s3.addfield("k", k3, description="Spring Constant", constant=True)
sim.s3.addfield("l", l3, description="Length", constant=True)
```

We now have to tell `simframe` in what order to update the fields.

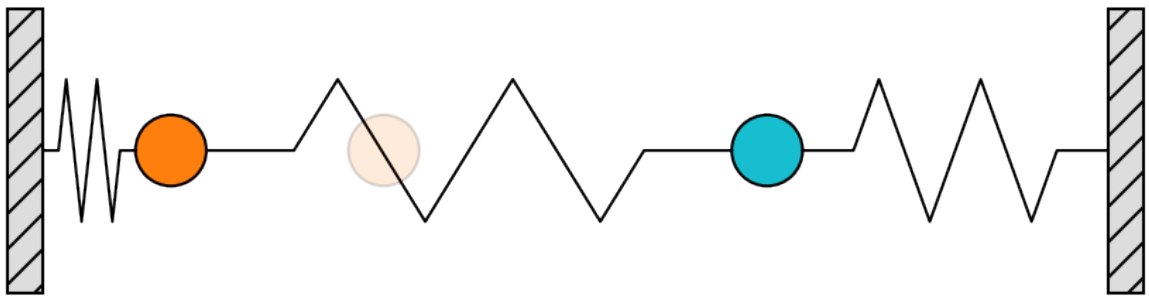
[21]:
```
# The groups for the bodies. The order does not matter
sim.updater = ["b1", "b2"]
# The fields in the groups. Displacement has to be updated before position
sim.b1.updater = ["d", "v", "x"]
sim.b2.updater = ["d", "v", "x"]
```

We can now fill the fields with their initial conditions from the state vector.

[22]:
```
sim.update()
```

The initial state of the system looks as follows:

[23]:
```
fig, ax = plot_system(0.5, sim.b1.x, sim.b2.x, L)
circ = patches.Circle((sim.b1.x0, 0.), 0.5, linewidth=1, edgecolor="#000000",␣
↪facecolor="C1", alpha=0.15)
ax.add_patch(circ)
plt.show()
```



Printing the complete frame structure

[24]:
```
sim.toc
```

```
Frame (Coupled Springs)
    - b1: Group (Body 1)
        - d: Field (Displacement)
        - m: Field (Mass), constant
        - v: Field (Velocity)
        - x: Field (Position)
        - x0: Field (Equilibrium Position), constant
    - b2: Group (Body 2)
        - d: Field (Displacement)
        - m: Field (Mass), constant
        - v: Field (Velocity)
        - x: Field (Position)
        - x0: Field (Equilibrium Position), constant
    - s1: Group (Spring 1)
        - k: Field (Spring Constant), constant
        - l: Field (Length), constant
    - s2: Group (Spring 2)
        - k: Field (Spring Constant), constant
        - l: Field (Length), constant
    - s3: Group (Spring 3)
        - k: Field (Spring Constant), constant
        - l: Field (Length), constant
    - t: Field (Time), Integration variable
    - Y: Field (State vector)
```

Setting up the Jacobian

For implicit schemes we have to calculate the Jacobian. Since in this case the Jacobian is constant with time, we can define it outside of the frame object.

[25]:
```
jac = np.array([[                              0.,                                0., 1., 0.
↪],
                [                              0.,                                0., 0., 1.
↪],
                [-(sim.s1.k+sim.s2.k)/sim.b1.m,              sim.s2.k/sim.b1.m, 0., 0.
↪],
                [             sim.s2.k/sim.b2.m, -(sim.s2.k+sim.s3.k)/sim.b2.m, 0., 0.
↪]], dtype=np.float)
```

[26]:
```
jac
```

[26]:
```
array([[ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.],
       [-3.,  2.,  0.,  0.],
       [ 2., -3.,  0.,  0.]])
```

[27]:
```
def jac_impl(sim, x):
    return jac
```

[28]:
```
sim.Y.jacobinator = jac_impl
```

Setting up the Integrator

We can now set up the integrator just as in the previous examples.

[29]:
```
from simframe import Integrator
from simframe import Instruction
from simframe import schemes
```

[30]:
```
sim.integrator = Integrator(sim.t)
```

[31]:
```
sim.integrator.instructions = [Instruction(schemes.impl_1_euler_direct, sim.Y)]
```

Setting up the Writer

We also have to set up the writer. In this case we don't want to write data files. So we simply write the data into a namespace

[32]:
```
from simframe import writers
```

[33]:
```
sim.writer = writers.namespacewriter
sim.writer.dumping = False
sim.writer.verbosity = 0
```

Starting the simulation

[34]:
```
sim.run()
```

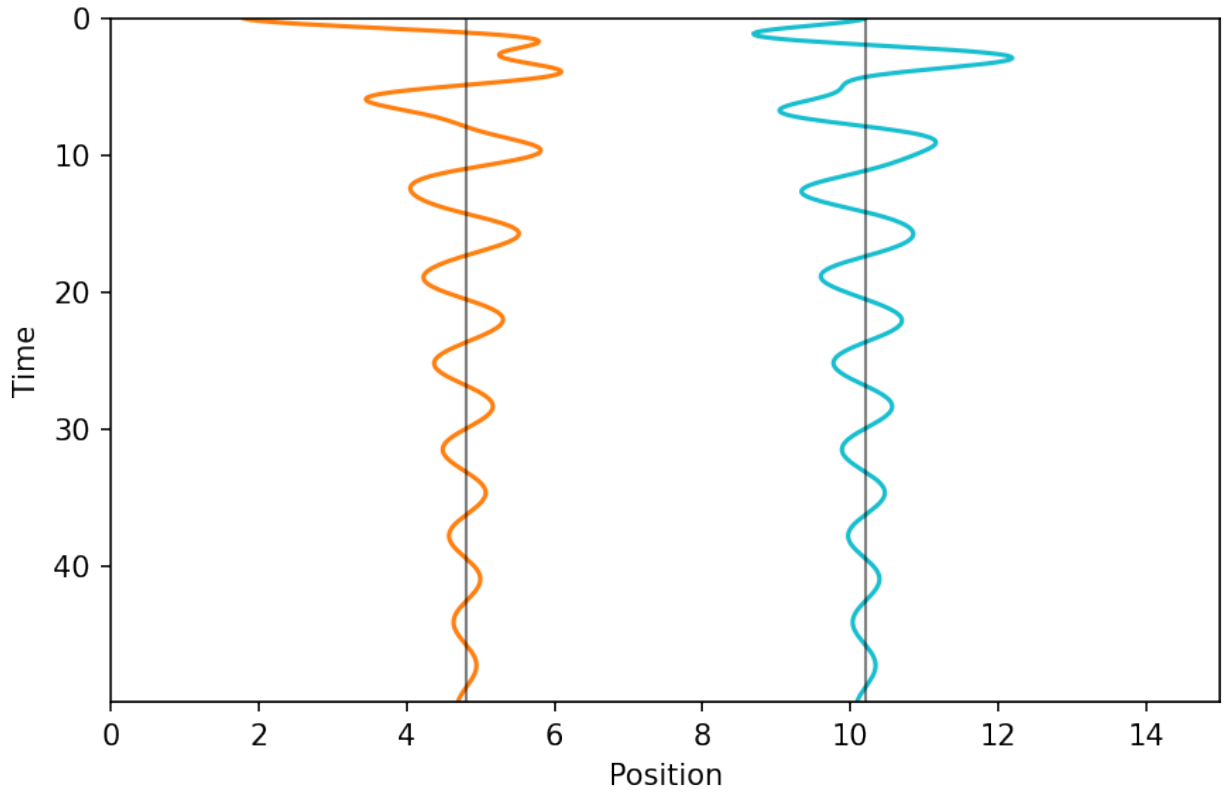
```
Execution time: 0:00:02
```

Reading data

[35]:
```
data = sim.writer.read.all()
```

[36]:
```
def plot_oszillations(data):
    fig, ax = plt.subplots(dpi=150)
    ax.plot(data.b1.x, data.t, c="C1")
    ax.plot(data.b2.x, data.t, c="C9")
    ax.axvline(data.b1.x0[0], c="#000000", alpha=0.5, lw=1)
    ax.axvline(data.b2.x0[0], c="#000000", alpha=0.5, lw=1)
    ax.set_xlim(0, L)
    ax.set_ylim(data.t[-1], data.t[0])
    ax.set_xlabel("Position")
    ax.set_ylabel("Time")
    fig.tight_layout()
```

[37]:
```
plot_oszillations(data)
```



[38]:
```
from matplotlib import animation
from IPython.display import HTML
```

[39]:
```
def plot_animation(data, bw):
    fig, ax = plt.subplots()
    ax.axis("off")
    ax.set_aspect(1.)
    l1, = ax.plot(data.b1.x, data.t-data.t[0], c="C1", lw=2, zorder=-1)
    l2, = ax.plot(data.b2.x, data.t-data.t[0], c="C9", lw=2, zorder=-1)
    rectl = patches.Rectangle((-bw, -4.*bw), bw, 8.*bw, linewidth=1, edgecolor="
↪#000000", facecolor="#dddddd", hatch="//")
    ax.add_patch(rectl)
    rectr = patches.Rectangle((L, -4.*bw), bw, 8.*bw, linewidth=1, edgecolor="#000000
↪", facecolor="#dddddd", hatch="//")
    ax.add_patch(rectr)
    ax.set_xlim(-2.*bw, L+2.*bw)
    ax.set_ylim(-5., 5.)
    b1 = patches.Circle((data.b1.x[0], 0), bw, linewidth=1, edgecolor="#000000",␣
↪facecolor="C1")
    ax.add_patch(b1)
    b2 = patches.Circle((data.b2.x[0], 0), bw, linewidth=1, edgecolor="#000000",␣
↪facecolor="C9")
    ax.add_patch(b2)
    x, y = getspring(0., data.b1.x[0]-bw, bw)
    s1, = ax.plot(x, y, c="#000000", lw=1)
    x, y = getspring(data.b1.x[0]+bw, data.b2.x[0]-bw, bw)
    s2, = ax.plot(x, y, c="#000000", lw=1)
```

(continues on next page)

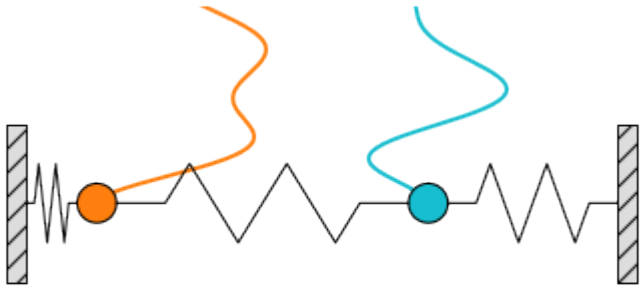
(continued from previous page)

```
    x, y = getspring(data.b2.x[0]+bw, L, bw)
    s3, = ax.plot(x, y, c="#000000", lw=1)
    return fig, ax, l1, l2, b1, b2, s1, s2, s3
```

[40]:
```
def init():
    l1.set_data(data.b1.x, data.t-data.t[0])
    l2.set_data(data.b2.x, data.t-data.t[0])
    b1.center = (data.b1.x[0], 0.)
    ax.add_patch(b1)
    b2.center = (data.b2.x[0], 0.)
    ax.add_patch(b2)
    x, y = getspring(0., data.b1.x[0]-bw, bw)
    s1.set_data(x, y)
    x, y = getspring(data.b1.x[0]+bw, data.b2.x[0]-bw, bw)
    s2.set_data(x, y)
    x, y = getspring(data.b2.x[0]+bw, L, bw)
    s3.set_data(x, y)
    return l1, l2, b1, b2, s1, s2, s3
```

[41]:
```
def animate(i):
    l1.set_data(data.b1.x, data.t-data.t[i])
    l2.set_data(data.b2.x, data.t-data.t[i])
    b1.center = (data.b1.x[i], 0.)
    b2.center = (data.b2.x[i], 0.)
    x, y = getspring(0., data.b1.x[i]-bw, bw)
    s1.set_data(x, y)
    x, y = getspring(data.b1.x[i]+bw, data.b2.x[i]-bw, bw)
    s2.set_data(x, y)
    x, y = getspring(data.b2.x[i]+bw, L, bw)
    s3.set_data(x, y)
    return l1, l2, b1, b2, s1, s2, s3
```

[42]:
```
fig, ax, l1, l2, b1, b2, s1, s2, s3 = plot_animation(data, 0.5)
```



[43]:
```
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=len(data.t), interval=30, blit=True)
```

[44]:
```
HTML(anim.to_html5_video())
```

[44]:
```
<IPython.core.display.HTML object>
```

As you can see the oscillation is damped pretty quickly, which is weird because we did not include any damping term into our differential equations for the velocities. The damping is purely numerically.

The cause for the damping is the implicit integrator scheme used here, that is not suited for the problem, similar to the explicit integrator used for the orbital integration.

The implicit midpoint method is simplectic, i.e., energy conserving. We can use this instead.

Resetting

[45]:
```
sim.Y = (-3., 0., 0., 0)
sim.update()
sim.t = 0
sim.writer.reset()
```

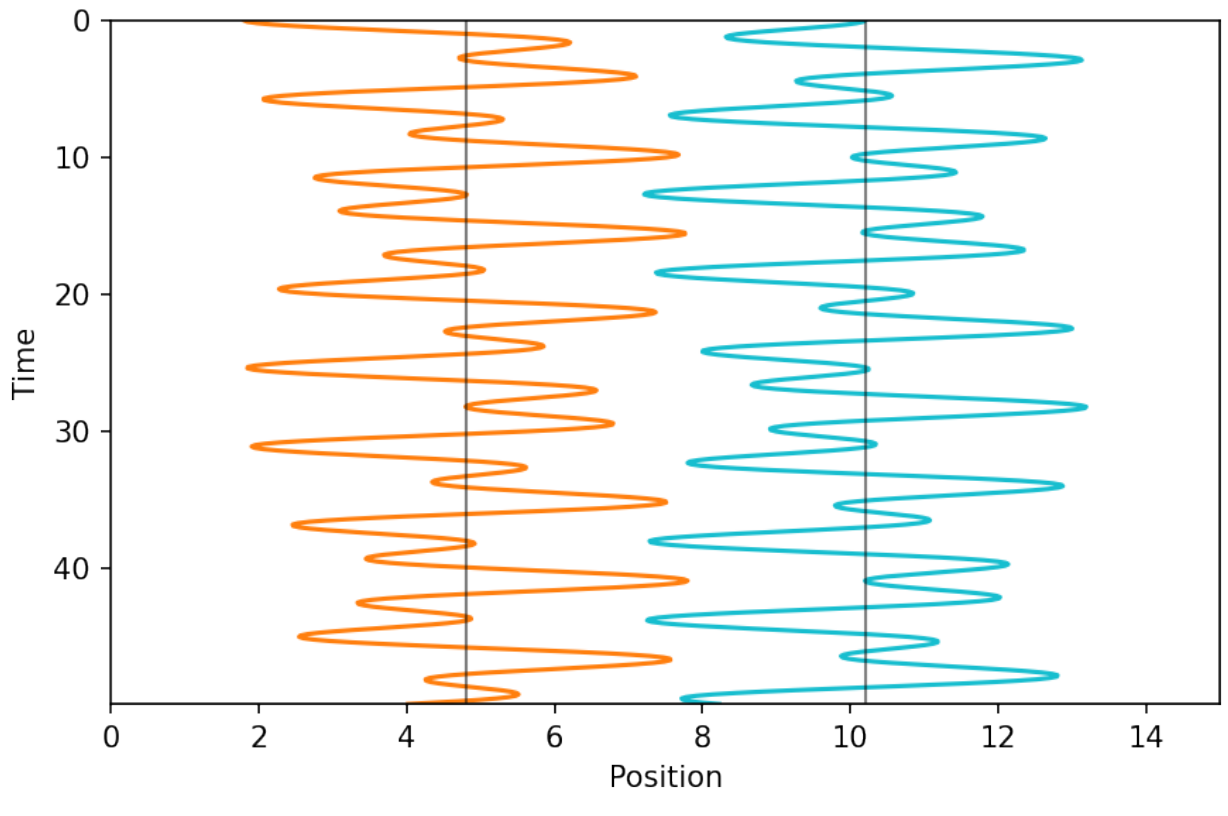
[46]:
```
sim.integrator.instructions = [Instruction(schemes.impl_2_midpoint_direct, sim.Y)]
```

[47]:
```
sim.run()
```

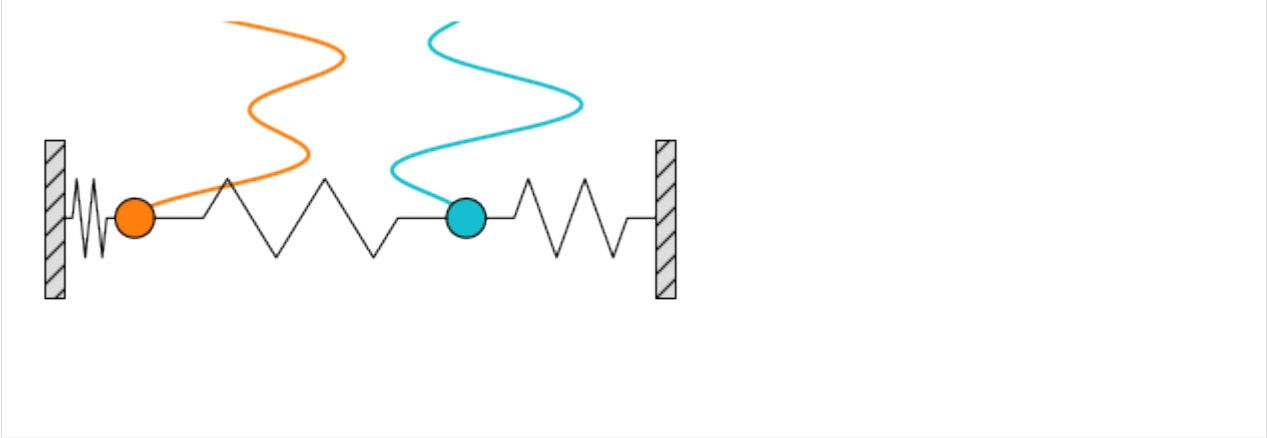
```
Execution time: 0:00:03
```

[48]:
```
data = sim.writer.read.all()
```

[49]:
```
plot_oszillations(data)
```



[50]:
```
fig, ax, l1, l2, b1, b2, s1, s2, s3 = plot_animation(data, 0.5)
```



[51]:
```
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=len(data.t), interval=30, blit=True)
```

[52]:
```
HTML(anim.to_html5_video())
```

[52]:
```
<IPython.core.display.HTML object>
```

Now the damping is gone.

But we could also easily use an explicit scheme.

For explicit integration, we do not have to set the differentiator, since we have set a Jacobian and `simframe` is automatically calculating the derivative from the Jacobian.

Resetting

[53]:
```
sim.Y = (-3., 0., 0., 0)
sim.update()
sim.t = 0
sim.writer.reset()
```

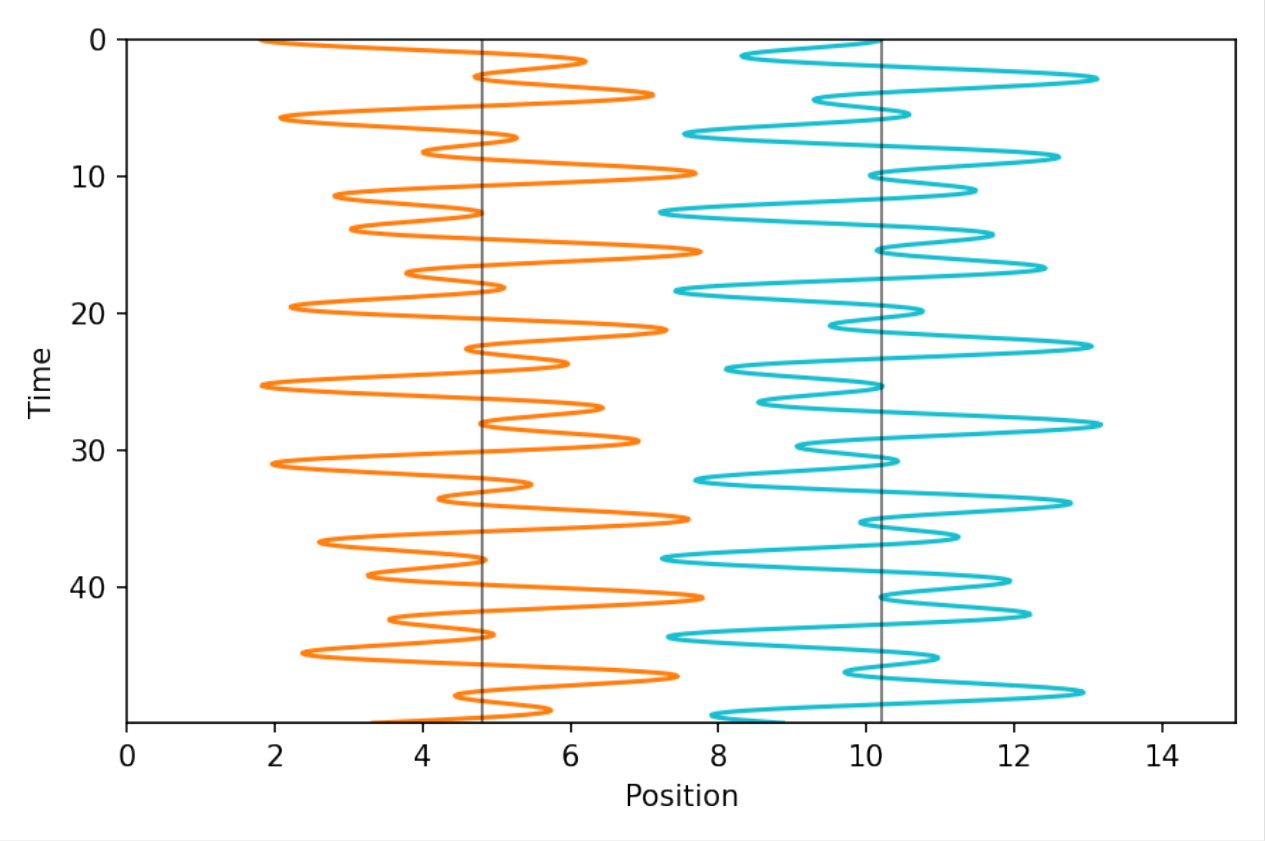
[54]:
```
sim.integrator.instructions = [Instruction(schemes.expl_4_runge_kutta, sim.Y)]
```

[55]:
```
sim.run()
```

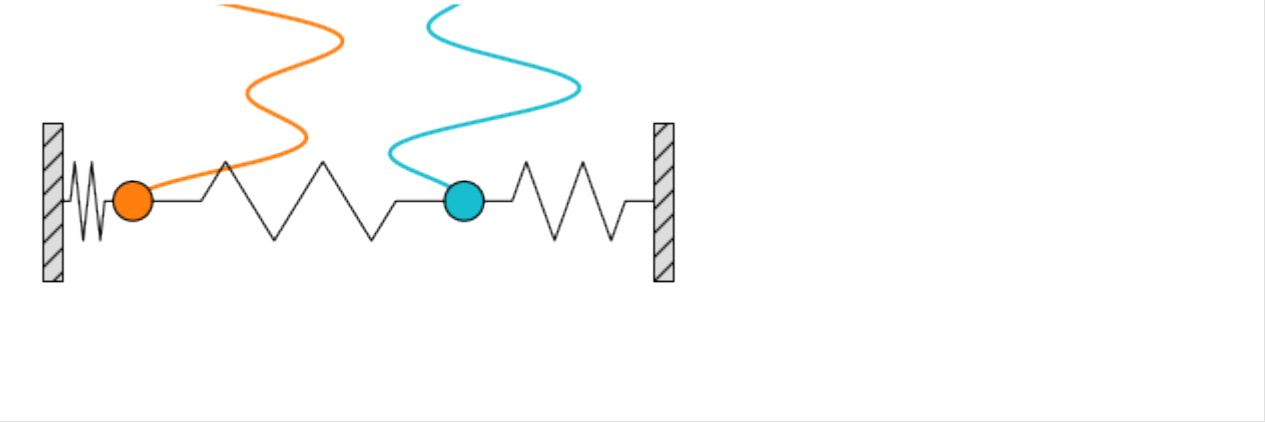
```
Execution time: 0:00:03
```

[56]:
```
data = sim.writer.read.all()
```

[57]:
```
plot_oszillations(data)
```



[58]:
```
fig, ax, l1, l2, b1, b2, s1, s2, s3 = plot_animation(data, 0.5)
```



[59]:
```
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=len(data.t), interval=30, blit=True)
```

[60]:
```
HTML(anim.to_html5_video())
```

[60]:
```
<IPython.core.display.HTML object>
```

CHAPTER

EIGHT

MODULE REFERENCE

8.1 simframe.frame Package

This package contains the core infrastructure of `simframe`.

8.1.1 Classes

AbstractGroup()	This is an abstract class that serves as template for other classes.
Field(owner, value[, updater, ...])	Class for storing simulation quantities.
Frame([integrator, writer, updater, ...])	This is the parent object of type `Group` that contains all other objects.
Group(owner[, updater, description])	Class for grouping data.
Heartbeat([updater, systole, diastole])	This class controls an update including `systole` and `diastole`.
IntVar(owner[, value, snapshots, updater, ...])	Cclass for integration variables that behaves as `Field` but has additional functionality with respect to stepsize management for integration.
Updater([func])	Class that manages how a `Group` or `Field` is updated.

AbstractGroup

class simframe.frame.**AbstractGroup**

Bases: `object`

This is an abstract class that serves as template for other classes.

`AbstractGroup` has a descriptive string, an owner and an updater. The owner is the parent `Frame` object and is hidden. The updater is of type `Heartbeat`.

`AbstractGroup` has an `update` method that is calling `systole`, `updater`, and `diastole` of the `Heartbeat` object, which manages the update of `AbstractGroup`.

`AbstractGroup` should not be instanciated directly.

Attributes

description Description of the instance.

updater `Heatbeat` object with instructions for updating the instance.

Methods

update(*args, **kwargs)	Function to update the object.

Attributes Summary

description	Description of the instance.
updater	`Heatbeat` object with instructions for updating the instance.

Methods Summary

update(*args, **kwargs)	Function to update the object.

Attributes Documentation

description
Description of the instance.

updater
`Heatbeat` object with instructions for updating the instance.

Methods Documentation

update(**args*, ***kwargs*)
Function to update the object. Ths functions calls the heartbeat instance of the object.

Parameters

- **args** (*additional positional arguments*) –
- **kwargs** (*additional keyword arguments*) –

Notes

Positional arguments and keyword arguments are only passed to the `updater`, NOT to `systole` and `diastole`.

Field

class simframe.frame.**Field**(*owner*, *value*, *updater=None*, *differentiator=None*, *jacobinator=None*, *description=''*, *constant=False*, *save=True*, *copy=False*)
Bases: `numpy.ndarray`, `simframe.frame.abstractgroup.AbstractGroup`

Class for storing simulation quantities.

In addition to `Group`, `Field` can have an `differentiator` for calculating its derivative and/or an `jacobinator` for calculating its Jacobian. The function that is calculating the derivative needs the parent `Frame` object as first, the integration variable of type `IntVar` as second, and the `Field` itself as third positional argument

`Field` behaves like `numpy.ndarray` and can perform the same numerical operations.

Notes

When `Field.update()` is called `Field` will be updated according return value of the `updater` of the `Heartbeat` object assigned to the `Field`. The function that is updating `Field` needs the parent `Frame` object as first positional argument.

Attributes

`T` The transposed array.

`base` Base object if memory is from some other object.

`buffer` Temporary buffer that stores the new value of `Field` after successful integration.

`constant` If True, `Field` is immutable.

`ctypes` An object to simplify the interaction of the array with the ctypes module.

`data` Python buffer object pointing to the start of the array's data.

`description` Description of the instance.

`differentiator` `Heartbeat` object with instructions for calculating the derivative of `Field`

`dtype` Data-type of the array's elements.

`flags` Information about the memory layout of the array.

`flat` A 1-D iterator over the array.

`imag` The imaginary part of the array.

`itemsize` Length of one array element in bytes.

`jacobinator` `Heartbeat` object with instructions for calculating the Jacobian of `Field`

`nbytes` Total bytes consumed by the elements of the array.

`ndim` Number of array dimensions.

`real` The real part of the array.

`save` If False, `Field` will not be stored in output files.

`shape` Tuple of array dimensions.

`size` Number of elements in the array.

`strides` Tuple of bytes to step in each dimension when traversing an array.

`updater` `Heatbeat` object with instructions for updating the instance.

Methods

all([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
any([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.
argmax([axis, out])	Return indices of the maximum values along the given axis.
argmin([axis, out])	Return indices of the minimum values along the given axis.
argpartition(kth[, axis, kind, order])	Returns the indices that would partition this array.
argsort([axis, kind, order])	Returns the indices that would sort this array.
astype(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
byteswap([inplace])	Swap the bytes of the array elements
choose(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
clip([min, max, out])	Return an array whose values are limited to [min, max].
compress(condition[, axis, out])	Return selected slices of this array along given axis.
conj()	Complex-conjugate all elements.
conjugate()	Return the complex conjugate, element-wise.
copy([order])	Return a copy of the array.
cumprod([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
cumsum([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
derivative([x, Y])	If differentiator or jacobinator is set, this returns the derivative of the Field.
diagonal([offset, axis1, axis2])	Return specified diagonals.
dot(b[, out])	Dot product of two arrays.
dump(file)	Dump a pickle of the array to the specified file.
dumps()	Returns the pickle of the array as a string.
fill(value)	Fill the array with a scalar value.
flatten([order])	Return a copy of the array collapsed into one dimension.
getfield(dtype[, offset])	Returns a field of the given array as a certain type.
item(*args)	Copy an element of an array to a standard Python scalar and return it.
itemset(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
jacobian([x])	If jacobinator is set, this returns the Jacobi matrix of the Field.
max([axis, out, keepdims, initial, where])	Return the maximum along a given axis.
mean([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
min([axis, out, keepdims, initial, where])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.

continues on next page

Table 5 – continued from previous page

prod([axis, dtype, out, keepdims, initial, ...])	Return the product of the array elements over the given axis
ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.
sort([axis, kind, order])	Sort an array in-place.
squeeze([axis])	Remove axes of length one from *a*.
std([axis, dtype, out, ddof, keepdims, where])	Returns the standard deviation of the array elements along given axis.
sum([axis, dtype, out, keepdims, initial, where])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
tostring([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
transpose(*axes)	Returns a view of the array with axes transposed.
update(*args, **kwargs)	Function to update the `Field`.
var([axis, dtype, out, ddof, keepdims, where])	Returns the variance of the array elements, along given axis.
view([dtype][, type])	New view of array with the same data.

Attributes Summary

buffer	Temporary buffer that stores the new value of Field after successful integration.
constant	If True, Field is immutable.
differentiator	Heartbeat object with instructions for calculating the derivative of Field
jacobinator	Heartbeat object with instructions for calculating the Jacobian of Field
save	If False, Field will not be stored in output files.

Methods Summary

derivative([x, Y])	If differentiator or jacobinator is set, this returns the derivative of the Field.
jacobian([x])	If jacobinator is set, this returns the Jacobi matrix of the Field.
update(*args, **kwargs)	Function to update the Field.

Attributes Documentation

buffer
Temporary buffer that stores the new value of Field after successful integration.

constant
If True, Field is immutable.

differentiator
Heartbeat object with instructions for calculating the derivative of Field

jacobinator
Heartbeat object with instructions for calculating the Jacobian of Field

save
If False, Field will not be stored in output files.

Methods Documentation

derivative (*x=None*, *Y=None*, **args*, ***kwargs*)
If differentiator or jacobinator is set, this returns the derivative of the Field.

Parameters

- **x** (IntVar, *optional, default : None*) – Integration variable If None it uses the integration variable of the integrator of the parent Frame
- **Y** (Field, *optional, default : None*) – Derivative of Y with respect to the integration variable. If None it uses the field itself

Returns deriv

Return type derivative of the field according the differetiator or jacobinator

Notes

The function that calculates the derivative needs the parent `Frame` as first positional, the integration variable `IntVar` as second positional, and the `Field` itself as third positional argument.

The `differentiator` is not set, it will try to calculate the derivative from the Jacobian. If `jacobinator` is also not set, it will return `False`

jacobian(*x=None*, **args*, ***kwargs*)
If `jacobinator` is set, this returns the Jacobi matrix of the `Field`.

Parameters **x** (`IntVar, optional, default : None`) – Integration variable If None it uses the integration variable of the integrator of the parent Frame

Returns

- **jac** (*Jacobi matrix of the field according the differetiator*)
- *The function that calculates the Jacobian needs the parent frame as first positional and the*
- *integration variable as second positional.*

update(**args*, ***kwargs*)
Function to update the `Field`.

Frame

class simframe.frame.**Frame**(*integrator=None*, *writer=None*, *updater=None*, *verbosity=2*, *progressbar=None*, *description=''*)
Bases: `simframe.frame.group.Group`

This is the parent object of type `Group` that contains all other objects.

During integration the `update()` function of the `Frame` object will be called. You have to sub-delegete the updates of your other `Group` and `Field` objects within this function.

`Frame` has additional functionality for writing output files and for integration.

Attributes

description Description of the instance.

integrator `Integrator` that controls the simulation.

progressbar `Progressbar` for displaying current status.

toc Complete table of contents starting from this object.

updateorder Update order if updater was set with list of strings.

updater `Heartbeat` object with update instructions.

verbosity Verbosity of the `Frame` objects.

writer `Writer` object for writing output files.

Methods

addfield(name, value[, updater, ...])	Function to add a new Field to the object.
addgroup(name[, updater, description])	Function to add a new Group to the object.
addintegrationvariable(name, value[, ...])	Function to add a new integration variable IntVar to the object.
run()	This method starts the simulation.
update(*args, **kwargs)	Function to update the object.
writeoutput([i, forceoverwrite, filename])	Writes output to file, if Writer is specified.

Attributes Summary

integrator	Integrator that controls the simulation.
progressbar	Progressbar for displaying current status.
verbosity	Verbosity of the Frame objects.
writer	Writer object for writing output files.

Methods Summary

run()	This method starts the simulation.
writeoutput([i, forceoverwrite, filename])	Writes output to file, if Writer is specified.

Attributes Documentation

integrator
Integrator that controls the simulation.

progressbar
Progressbar for displaying current status.

verbosity
Verbosity of the Frame objects.

writer
Writer object for writing output files.

Methods Documentation

run()
This method starts the simulation. An Integrator has to be set beforehand.

writeoutput(*i=0*, *forceoverwrite=False*, *filename=''*, ***kwargs*)
Writes output to file, if Writer is specified.

Parameters

- **i** (*int, optional, default : 0*) – Number of output
- **forceoverwrite** (*boolean, optional, default : False*) – If True, this overrules the seetings of the Writer and enforces the file to be overwritten.

- **filename** (*string, optional, default = ""*) – if given this will write the output to this file. Otherwise, it uses the standard scheme.
- **kwargs** (*additional keyword arguments*) – Additional options that can be passed to the writer

Group

class simframe.frame.**Group**(*owner*, *updater=None*, *description=''*)
Bases: simframe.frame.abstractgroup.AbstractGroup

Class for grouping data. Group is a data frame that has additional functionality for updating its attributes.

Notes

When Group.update() is called the instructions of the group's Heartbeat object will be performed. The function that is determing the update operation needs the parent Frame object as first positional argument.

Attributes

description Description of the instance.

toc Complete table of contents starting from this object.

updateorder Update order if updater was set with list of strings.

updater Heartbeat object with update instructions.

Methods

addfield(name, value[, updater, ...])	Function to add a new Field to the object.
addgroup(name[, updater, description])	Function to add a new Group to the object.
addintegrationvariable(name, value[, ...])	Function to add a new integration variable IntVar to the object.
update(*args, **kwargs)	Function to update the object.

Attributes Summary

toc	Complete table of contents starting from this object.
updateorder	Update order if updater was set with list of strings.
updater	Heartbeat object with update instructions.

Methods Summary

addfield(name, value[, updater, ...])	Function to add a new `Field` to the object.
addgroup(name[, updater, description])	Function to add a new `Group` to the object.
addintegrationvariable(name, value[, ...])	Function to add a new integration variable `IntVar` to the object.

Attributes Documentation

toc
Complete table of contents starting from this object.

updateorder
Update order if updater was set with list of strings. `None` otherwise.

updater
`Heartbeat` object with update instructions.

You can either set a `Heartbeat` object directly, a callable functions that will be automatically transformed into a `Heartbeat` object, or a list of attribute names of the `Group` that will be updated in that order.

Methods Documentation

addfield(*name*, *value*, *updater=None*, *differentiator=None*, *description=''*, *constant=False*, *copy=True*)
Function to add a new `Field` to the object.

Parameters

- **name** (*string*) – Name of the field
- **value** (*number, array, string*) – Initial value of the field. Needs to have already the correct type and shape
- **updater** (*Heartbeat, Updater, callable or None, optional, default : None*) – Updater for field update
- **differentiator** (*Heartbeat, Updater, callable or None, optional, default : None*) – Differentiator if the field has a derivative
- **description** (*string, optional, default : ""*) – Descriptive string for the field
- **constant** (*boolean, optional, default : False*) – True if the field is immutable
- **copy** (*boolean, optional, default : True*) – If True <value> will be copied, not referenced

addgroup(*name*, *updater=None*, *description=''*)
Function to add a new `Group` to the object.

Parameters

- **name** (*string*) – Name of the group
- **updater** (*Heartbeat, Updater, callable or None, optional, default : None*) – Updater for field update

- **description** (*string, optional, default : ""*) – Descriptive string for the group

addintegrationvariable(*name*, *value*, *snapshots=[]*, *updater=None*, *description=''*, *copy=True*)

Function to add a new integration variable `IntVar` to the object.

Parameters

- **name** (*string*) – Name of the field
- **value** (*number, array, string*) – Initial value of the field. Needs to have already the correct type and shape
- **updater** (`Heartbeat`, `Updater`, *callable or None, optional, default : None*) – Updater for field update
- **snapshots** (*list, ndarray, optional, default : []*) – List of snapshots at which an output file should be written
- **description** (*string, optional, default : ""*) – Descriptive string for the field
- **copy** (*boolean, optional, default : True*) – If True <value> will be copied, not referenced

Heartbeat

class simframe.frame.**Heartbeat**(*updater=None*, *systole=None*, *diastole=None*)

Bases: `object`

This class controls an update including `systole` and `diastole`.

A full cardiac cycle constists of a systole operation, the actual update and a systole operation. All three are of type `Updater`.

The `beat` function calls `systole`, `updater`, `diastole` in that order and returns the return value of the `updater`. Any positional or keyword arguments are only passed to the `updater`, NOT to `systole` and `diastole`.

Attributes

diastole `Updater` that is called at the end of the cardiac cycle.

systole `Updater` that is called in the beginning of cardiac cycle.

updater `Updater` that is performing the update instruction.

Methods

beat(owner, *args[, Y])	This method executes `systole`, `updater`, and `distole` in that order and returns the return value of the `updater`.

Attributes Summary

diastole	Updater that is called at the end of the cardiac cycle.
systole	Updater that is called in the beginning of cardiac cycle.
updater	Updater that is performing the update instruction.

Methods Summary

beat(owner, *args[, Y])	This method executes systole, updater, and distole in that order and returns the return value of the updater.

Attributes Documentation

diastole
Updater that is called at the end of the cardiac cycle.

systole
Updater that is called in the beginning of cardiac cycle.

updater
Updater that is performing the update instruction.

Methods Documentation

beat (*owner*, **args*, *Y=None*, ***kwargs*)
This method executes systole, updater, and distole in that order and returns the return value of the updater.

Parameters

- **owner** (Frame) – Parent frame object to which updater belongs
- **Y** (Field, *optional, default : None*) – Field that should be updated

Notes

*args and **kwargs are only passed to updater, NOT to systole and diastole

Returns **ret**

Return type Return value of updater.

IntVar

class simframe.frame.**IntVar**(*owner*, *value=0*, *snapshots=[]*, *updater=None*, *description=''*, *save=True*, *copy=False*)

Bases: simframe.frame.field.Field

Cclass for integration variables that behaves as Field but has additional functionality with respect to stepsize management for integration.

Notes

The updater for integration variables is calculating the stepsize. The function associated to the updater needs the parent Frame object as first positional argument and needs to return the desired stepsize.

IntVar.update() does not update the integration variable. Try not to update the integration variable by hand. Let the Integrator do it for you.

Attributes

T The transposed array.

base Base object if memory is from some other object.

buffer Temporary buffer that stores the new value of Field after successful integration.

constant If True, Field is immutable.

ctypes An object to simplify the interaction of the array with the ctypes module.

data Python buffer object pointing to the start of the array's data.

description Description of the instance.

differentiator Heartbeat object with instructions for calculating the derivative of Field

dtype Data-type of the array's elements.

flags Information about the memory layout of the array.

flat A 1-D iterator over the array.

imag The imaginary part of the array.

itemsize Length of one array element in bytes.

jacobinator Heartbeat object with instructions for calculating the Jacobian of Field

maxstepsize Maximum possible step size, i.e., to next snapshot.

nbytes Total bytes consumed by the elements of the array.

ndim Number of array dimensions.

nextsnapshot Value of the next snapshot.

prevsnapshot Value of the previous snapshot.

prevstepsize Previously taken step size.

real The real part of the array.

save If False, Field will not be stored in output files.

shape Tuple of array dimensions.

size Number of elements in the array.

snapshots Snapshots at which output should be written.

stepsize Current stepsize.

strides Tuple of bytes to step in each dimension when traversing an array.

suggested Suggested step size.

updater `Heatbeat` object with instructions for updating the instance.

Methods

`all`([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
`any`([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.
`argmax`([axis, out])	Return indices of the maximum values along the given axis.
`argmin`([axis, out])	Return indices of the minimum values along the given axis.
`argpartition`(kth[, axis, kind, order])	Returns the indices that would partition this array.
`argsort`([axis, kind, order])	Returns the indices that would sort this array.
`astype`(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
`byteswap`([inplace])	Swap the bytes of the array elements
`choose`(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
`clip`([min, max, out])	Return an array whose values are limited to `[min, max]`.
`compress`(condition[, axis, out])	Return selected slices of this array along given axis.
`conj`()	Complex-conjugate all elements.
`conjugate`()	Return the complex conjugate, element-wise.
`copy`([order])	Return a copy of the array.
`cumprod`([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
`cumsum`([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
`derivative`([x, Y])	If `differentiator` or `jacobinator` is set, this returns the derivative of the `Field`.
`diagonal`([offset, axis1, axis2])	Return specified diagonals.
`dot`(b[, out])	Dot product of two arrays.
`dump`(file)	Dump a pickle of the array to the specified file.
`dumps`()	Returns the pickle of the array as a string.
`fill`(value)	Fill the array with a scalar value.
`flatten`([order])	Return a copy of the array collapsed into one dimension.
`getfield`(dtype[, offset])	Returns a field of the given array as a certain type.
`item`(*args)	Copy an element of an array to a standard Python scalar and return it.
`itemset`(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
`jacobian`([x])	If `jacobinator` is set, this returns the Jacobi matrix of the `Field`.
`max`([axis, out, keepdims, initial, where])	Return the maximum along a given axis.

continues on next page

Table 17 – continued from previous page

mean([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
min([axis, out, keepdims, initial, where])	Return the minimum along a given axis.
newbyteorder([new_order])	Return the array with the same data viewed with a different byte order.
nonzero()	Return the indices of the elements that are non-zero.
partition(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
prod([axis, dtype, out, keepdims, initial, ...])	Return the product of the array elements over the given axis
ptp([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
put(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
ravel([order])	Return a flattened array.
repeat(repeats[, axis])	Repeat elements of an array.
reshape(shape[, order])	Returns an array containing the same data with a new shape.
resize(new_shape[, refcheck])	Change shape and size of array in-place.
round([decimals, out])	Return *a* with each element rounded to the given number of decimals.
searchsorted(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
setfield(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
setflags([write, align, uic])	Set array flags WRITEABLE, ALIGNED, (WRITEBACKIFCOPY and UPDATEIFCOPY), respectively.
sort([axis, kind, order])	Sort an array in-place.
squeeze([axis])	Remove axes of length one from *a*.
std([axis, dtype, out, ddof, keepdims, where])	Returns the standard deviation of the array elements along given axis.
suggest(value[, reset])	Suggest a step size
sum([axis, dtype, out, keepdims, initial, where])	Return the sum of the array elements over the given axis.
swapaxes(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
take(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
tobytes([order])	Construct Python bytes containing the raw data bytes in the array.
tofile(fid[, sep, format])	Write array to a file as text or binary (default).
tolist()	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
tostring([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
trace([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
transpose(*axes)	Returns a view of the array with axes transposed.
update()	Not used for `IntVar`.

continues on next page

Table 17 – continued from previous page

var([axis, dtype, out, ddof, keepdims, where])	Returns the variance of the array elements, along given axis.
view([dtype][, type])	New view of array with the same data.

Attributes Summary

maxstepsize	Maximum possible step size, i.e., to next snapshot.
nextsnapshot	Value of the next snapshot.
prevsnapshot	Value of the previous snapshot.
prevstepsize	Previously taken step size.
snapshots	Snapshots at which output should be written.
stepsize	Current stepsize.
suggested	Suggested step size.

Methods Summary

suggest(value[, reset])	Suggest a step size
update()	Not used for IntVar.

Attributes Documentation

maxstepsize
Maximum possible step size, i.e., to next snapshot.

nextsnapshot
Value of the next snapshot.

prevsnapshot
Value of the previous snapshot.

prevstepsize
Previously taken step size.

snapshots
Snapshots at which output should be written.

Even if no outputs are written it needs to contain at least one value that specifies the end point of the simulation.

stepsize
Current stepsize.

suggested
Suggested step size.

Methods Documentation

suggest(*value*, *reset=False*)
Suggest a step size

For adaptive integration schemes, this function can be used to suggest a step size for the next integration step. If many vaiables are integrated this safes the smallest suggested step size in a temporary buffer accessible via `IntVar.suggested`.

Parameters

- **value** (`Field`) – Suggested step size
- **reset** (*boolean, optional, default : False*) – If True, the previous value will be descarded.

update()
Not used for `IntVar`.

Updater

class simframe.frame.**Updater**(*func=None*)
Bases: `object`

Class that manages how a `Group` or `Field` is updated.

Methods

update(owner, *args, **kwargs)	Function that is called when `Group` or `Field` to which `Updater` belongs is being updated.

Methods Summary

update(owner, *args, **kwargs)	Function that is called when `Group` or `Field` to which `Updater` belongs is being updated.

Methods Documentation

update(*owner*, **args*, ***kwargs*)
Function that is called when `Group` or `Field` to which `Updater` belongs is being updated.

Parameters

- **owner** (`Frame`) – Parent `Frame` object
- **args** (*additional positional arguments*) –
- **kwargs** (*additonal keyword arguments*) –

8.2 simframe.integration Package

This package contains infrastructure for solving differential equations within `simframe`. The `Integrator` class is the basic class that advances the simulation from snapshot to snapshot by executing one integration `Instruction` at a time. Instructions contain a list of integration `Scheme`. The `schemes` package contains pre-defined integration schemes that are ready to use in `simframe`.

8.2.1 Classes

Instruction(scheme, Y[, fstep, controller, ...])	Integration `Instruction` that controls the execution of an integration `Scheme`.
Integrator(var[, instructions, failop, ...])	`Integrator` class that manages the integration instructions
Scheme(scheme[, controller, description])	Class for an integration `Scheme` that can be used as template for creating custom schemes.

Instruction

class simframe.integration.**Instruction**(*scheme*, *Y*, *fstep=1.0*, *controller={}*, *description=''*)

Bases: simframe.integration.scheme.Scheme

Integration `Instruction` that controls the execution of an integration `Scheme`.

Attributes

Y The `Field` to be integrated

controller Dictionary with keyword arguments that is passed to the integration `Scheme`.

description Description of the `Scheme`.

fstep Fraction of step size the `Scheme` will be applied to `Field`

scheme Integration `Scheme`.

Methods

__call__([dx])	Execution of the integration instruction

Attributes Summary

Y	The `Field` to be integrated
fstep	Fraction of step size the `Scheme` will be applied to `Field`

Methods Summary

`__call__`([dx])	Execution of the integration instruction

Attributes Documentation

Y
The `Field` to be integrated

fstep
Fraction of step size the `Scheme` will be applied to `Field`

Methods Documentation

__call__(*dx=None*)
Execution of the integration instruction

Parameters **dx** (*IntVar, optional, default : None*) – Stepsize of the integration variable

Returns **Y1** – New value of the variable to be integrated

Return type *Field*

Integrator

class simframe.integration.**Integrator**(*var*, *instructions=[]*, *failop=None*, *preparator=None*, *finalizer=None*, *maxit=500*, *description=''*)

Bases: `object`

`Integrator` class that manages the integration instructions

Attributes

description Description of integrator

failop `Heartbeat` objects that is called if any integration `Instruction` returned `False`

finalizer `Heartbeat` object that is called after the integration was successful.

instructions List of integration `Instructions` that will be executed in that order.

maxit Maximum number of integration tries until program will be aborted.

preparator `Heartbeat` object that is called before the integration instructions will be executed.

var The integration variable `IntVar` that is associated with this `Integrator`.

Methods

integrate()	Method that executes one integration step.

Attributes Summary

description	Description of integrator
failop	`Heartbeat` objects that is called if any integration `Instruction` returned `False`
finalizer	`Heartbeat` object that is called after the integration was successful.
instructions	List of integration `Instructions` that will be executed in that order.
maxit	Maximum number of integration tries until program will be aborted.
preparator	`Heartbeat` object that is called before the integration instructions will be executed.
var	The integration variable `IntVar` that is associated with this `Integrator`.

Methods Summary

integrate()	Method that executes one integration step.

Attributes Documentation

description
Description of integrator

failop
`Heartbeat` objects that is called if any integration `Instruction` returned `False`

finalizer
`Heartbeat` object that is called after the integration was successful.

instructions
List of integration `Instructions` that will be executed in that order.

maxit
Maximum number of integration tries until program will be aborted.

preparator
`Heartbeat` object that is called before the integration instructions will be executed.

var
The integration variable `IntVar` that is associated with this `Integrator`.

Methods Documentation

integrate()
Method that executes one integration step.

Scheme

class simframe.integration.**Scheme**(*scheme*, *controller={}*, *description=''*)
Bases: object

Class for an integration Scheme that can be used as template for creating custom schemes.

Notes

The integration Scheme needs to return False if the integration failed. The Integrator will then perform a fail operation and will try it again. This can be used to implement schemes with adaptive step sizes. If the step size was not small enough the fail operation can reduce it further.

Attributes

controller Dictionary with keyword arguments that is passed to the integration Scheme.

description Description of the Scheme.

scheme Integration Scheme.

Methods

__call__(x0, Y0, dx, *args, **kwargs)	Method for returning the new value of the variable to be integrated.

Attributes Summary

controller	Dictionary with keyword arguments that is passed to the integration Scheme.
description	Description of the Scheme.
scheme	Integration Scheme.

Methods Summary

__call__(x0, Y0, dx, *args, **kwargs)	Method for returning the new value of the variable to be integrated.

Attributes Documentation

controller
Dictionary with keyword arguments that is passed to the integration `Scheme`.

description
Description of the `Scheme`.

scheme
Integration `Scheme`.

Methods Documentation

__call__ (*x0*, *Y0*, *dx*, **args*, ***kwargs*)
Method for returning the new value of the variable to be integrated.

Parameters

- **x0** (*Intvar*) – Integration variable at beginning of scheme
- **Y0** (`Field`) – Variable to be integrated at the beginning of scheme
- **dx** (`IntVar`) – Stepsize of integration variable
- **controller** (*dict, optional, default : {}*) – Additional keyword arguments passed to integration scheme
- **args** (*additional positional arguments*) –
- **kwargs** (*additional kewyowrd arguments*) –

Returns **Y1** – New value of the variable to be integrated. Functions needs to return `False` if integration failed.

Return type *Field* or False

8.3 simframe.integration.schemes Package

This package contains pre-defined instances of integration schemes that can be used in `simframe`. The naming convention is `<expl/impl>_<order>_<name>(<_additonal>)`.

For example: The fifth-order adaptive Cash-Karp scheme is `expl_5_cash_karp_adptv`, the 1st-order implicit Euler scheme using a GMRES solver is `impl_1_euler_gmres`.

8.4 simframe.io Package

This package is for input/output operations. It contains template `Writer` and `Reader` classes that can be used to create customized writing and reading methods. The package `writers` contains pre-defined `Writer` instances for writing and reading `simframe` data. The package furthermore contains a method for reading dump files and for printing a progress bar in an interactive shell.

8.4.1 Functions

readdump(filename)	Reads dumpfile and returns `Frame` object

readdump

`simframe.io.`**`readdump`**(*filename*)

Reads dumpfile and returns `Frame` object

Parameters **`filename`** (*`str`*) – Path to file to be read

Returns **obj** – object read from dump file

Return type object

Notes

Only read dump files from sources you trust. Malware can be injected.

8.4.2 Classes

Reader(writer[, description])	General class for reading output files.
Writer(func[, datadir, filename, zfill, ...])	General class for writing output files.
Progressbar([prefix, suffix, fill, empty, ...])	Class for printing progress bar to terminal.

Reader

`class` `simframe.io.`**`Reader`**(*writer*, *description=''*)

Bases: `object`

General class for reading output files. Every `Writer` class should also provide a reader for its data files.

Custom `Reader` must provide a method `Reader.output()` that reads a single output file and returns the data set as type `SimpleNamespace`.

The general `Reader` class provides a function that stitches together all `SimpleNamespaces` the `Reader.output()` method provides into a single `SimpleNamespace` by adding another dimension along the integration variable `IntVar`.

Attributes

`description` Description of the `Reader`.

Methods

all()	Functions that reads all output files and combines them into a single `SimpleNamespace`.
listfiles()	Method to list all data files in a directory
output(file)	Function that returns the data of a single output file.
sequence(field)	Function that returns the entire sequence of a specific field.

Attributes Summary

description	Description of the `Reader`.

Methods Summary

all()	Functions that reads all output files and combines them into a single `SimpleNamespace`.
listfiles()	Method to list all data files in a directory
output(file)	Function that returns the data of a single output file.
sequence(field)	Function that returns the entire sequence of a specific field.

Attributes Documentation

description
Description of the `Reader`.

Methods Documentation

all()
Functions that reads all output files and combines them into a single `SimpleNamespace`.

Returns **dataset** – Namespace of data set.

Return type SimpleNamespace

Notes

This function is reading one output files to get the structure of the data and calls `read.sequence()` for every field in the data structure.

listfiles()
Method to list all data files in a directory

Returns **files** – List of strings of all found data files sorted alphanumerically.

Return type list

Notes

Function only searches for files that match the pattern specified by the `Writer`'s `filename` and `extension` attributes.

output (*file*)
Function that returns the data of a single output file.

Parameters **file** (*str*) – Path to file that should be read.

Returns **data** – Data set of a single output file.

Return type SimpleNamespace

sequence (*field*)
Function that returns the entire sequence of a specific field.

Parameters **field** (*str*) – String with location of requested field

Returns **seq** – Array with requested values

Return type array

Notes

`field` is addressing the values just as in the parent frame object. E.g. `"groupA.groupB.fieldC"` is addressing `Frame.groupA.groupB.fieldC`.

Writer

class simframe.io.**Writer**(*func*, *datadir='data'*, *filename='data'*, *zfill=4*, *extension='out'*, *overwrite=False*, *dumping=True*, *reader=None*, *verbosity=1*, *description=''*, *options={}*)

Bases: `object`

General class for writing output files. It should be used as wrapper for customized `Writer`.

Attributes

datadir Data directory of output files.

description Description of `Writer`.

dumping If `True` dump files will be written.

extension Filename extension of output files.

filename Base filename of output files.

options Dictionary of keyword arguments passed to customized writing routine.

overwrite If `True` existing output files will be overwritten.

read `Reader` object for reading output files.

verbosity Verbosity of the writer.

zfill Zero padding of numbered files names.

Methods

checkdatadir([datadir, createdir])	Function checks if data directory exists and creates it if necessary.
write(owner, i, forceoverwrite[, filename])	Writes output to file
writedump(frame[, filename])	Writes the `Frame` to dump file

Attributes Summary

datadir	Data directory of output files.
description	Description of `Writer`.
dumping	If `True` dump files will be written.
extension	Filename extension of output files.
filename	Base filename of output files.
options	Dictionary of keyword arguments passed to customized writing routine.
overwrite	If `True` existing output files will be overwritten.
read	`Reader` object for reading output files.
verbosity	Verbosity of the writer.
zfill	Zero padding of numbered files names.

Methods Summary

checkdatadir([datadir, createdir])	Function checks if data directory exists and creates it if necessary.
write(owner, i, forceoverwrite[, filename])	Writes output to file
writedump(frame[, filename])	Writes the `Frame` to dump file

Attributes Documentation

datadir
Data directory of output files.

description
Description of `Writer`.

dumping
If `True` dump files will be written.

extension
Filename extension of output files.

filename
Base filename of output files.

options
Dictionary of keyword arguments passed to customized writing routine.

overwrite
If `True` existing output files will be overwritten.

read
Reader object for reading output files.

verbosity
Verbosity of the writer.

zfill
Zero padding of numbered files names.

Methods Documentation

checkdatadir(*datadir=None*, *createdir=False*)
Function checks if data directory exists and creates it if necessary.

Parameters

- **datadir** (*string or None, optinal, default : None*) – Data directory to be checked. If None it assumes the data directory of the parent writer.
- **createdir** (*boolen, optional, default : False*) – If True function creates data directory if it does not exist.

Returns **datadirexists** – True if directory exists, False if not

Return type boolean

write(*owner*, *i*, *forceoverwrite*, *filename=''*)
Writes output to file

Parameters

- **owner** (Frame) – Parent Frame object
- **i** (*int*) – Number of output
- **forceoverwrite** (*boolean*) – If True it will forces and overwrite of the file if it exists.
- **filename** (*string*) – If this is not "" the writer will use this filename instead of the standard scheme

writedump(*frame*, *filename=''*)
Writes the Frame to dump file

Parameters

- **frame** (*object*) – object to be written to file
- **filename** (*str, optional, default : ''*) – path to file to be written if not set, filename will be <writer.datadir>/frame.dmp.

Progressbar

class simframe.io.**Progressbar**(*prefix=' '*, *suffix='| '*, *fill=''*, *empty=' '*, *length=25*, *color='blue'*, *spinner=None*)
Bases: object

Class for printing progress bar to terminal.

Methods

__call__(x, x0, x1, s0, s1)	Prints the current progress bar.
print(x, x0, x1, s0, s1)	Function prints the current progress bar.

Methods Summary

__call__(x, x0, x1, s0, s1)	Prints the current progress bar.
print(x, x0, x1, s0, s1)	Function prints the current progress bar.

Methods Documentation

__call__ (*x*, *x0*, *x1*, *s0*, *s1*)
Prints the current progress bar.

Parameters

- **x** (*Number*) – Current state of progress
- **x0** (*Number*) – Starting point of snapshot
- **x1** (*Number*) – End point of snapshot
- **s0** (*Number*) – Starting point of simulation
- **s1** (*Number*) – End point of simulation

print (*x*, *x0*, *x1*, *s0*, *s1*)
Function prints the current progress bar. If the end of either the snapshot or the simulation is reached, no progress bar will be printed.

Parameters

- **x** (*Number*) – Current state of progress
- **x0** (*Number*) – Starting point of snapshot
- **x1** (*Number*) – End point of snapshot
- **s0** (*Number*) – Starting point of simulation
- **s1** (*Number*) – End point of simulation

8.5 simframe.io.writers Package

This package contains pre-defined `Writer` instances that can be used for writing and reading `Frame` objects. The `hdf5writer` writes data files in the HDF5 file format. The `namespacewriter` does not write output files (except for dump files if required). The data is stored locally in the `Writer` object itself.

8.6 simframe.utils Package

Package contains utility classes that facilitate the use of `simframe`

`Color` is a generic class that can be used to colorize text. `colorize` is an instance of `Color`, that can be called to add decorators to a string for colored output.

8.6.1 Classes

Color([color])	Class to decorate strings with color tags.

Color

class simframe.utils.**Color**(*color='reset'*)

Bases: `object`

Class to decorate strings with color tags.

Attributes

color Of type `Color` with color information.

Methods

__call__(s[, color])	Colorizes string

Attributes Summary

color	Of type `Color` with color information.

Methods Summary

__call__(s[, color])	Colorizes string

Attributes Documentation

color

Of type `Color` with color information.

Methods Documentation

__call__(*s*, *color=None*)
Colorizes string

Parameters

- **s** (*string*) – String to be colorized
- **color** (*string, optional, default : None*) – Color used to colorize. If None, standard color is used

Returns **cstr** – Colorized string

Return type string

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Y

Z