triflow Documentation

Release 0.4.0

Nicolas Cellier

May 09, 2017

CONTENTS:

INSTALLATION

1.1 External requirements

The library is based on Theano, thus extra dependecies like fortran and C compiler are needed, see Theano install page for extra informations:

http://deeplearning.net/software/theano/install.html

1.2 via PyPI

Beware, the PyPI version is not always up-to-date.

```
pip install triflow
```

will install the package and

```
pip install triflow --upgrade
```

will update an old version of the library.

use sudo if needed, and the user flag if you want to install it without the root privileges:

```
pip install --user triflow
```

1.3 via github

You can install the last version of the library using pip and the github repository:

```
pip install git+git://github.com/locie/triflow.git
```

OVERVIEW

2.1 Motivation

The aim of this library is to have an easy way to write transient dynamic systems with 1D finite difference discretisation, with fast temporal solvers.

The main two parts of the library are:

- symbolic tools defining the spatial discretisation.

- a fast temporal solver written to use the sparsity of the finite difference method to reduce the memory and CPU usage during the computation. _Theano make this part easy.

Moreover, we provide extra tools and we write the library in a modular way, allowing an easy extension of these different parts (see the plug-in module of the library.)

The library fits well with an interactive usage (in a jupyter notebook).

2.2 Model writing

We write all the models as function generating the F vector and the Jacobian matrix of the model defined as

$$\frac{\partial U}{\partial t} = F(U)$$

We write the symbolic model as a simple mathematic equation. For exemple, a diffusion advection model:

```
>>> from triflow import Model

>>> eq_diff = "k * dxxU - c * dxU"
>>> dep_var = "U"
>>> pars = ["k", "c"]

>>> model = Model(eq_diff, dep_var, pars)
```

the model give us access after that to the compiled routines for F and the corresponding Jacobian matrix as:

```
>>> print(model.F)
Matrix([[-2*U*k/dx**2 + 0.5*U_m1*c/dx + U_m1*k/dx**2 - 0.5*U_p1*c/dx + U_p1*k/dx**2]])

>>> print(model.J)
Matrix([
[ 0.5*c/dx + k/dx**2],
[         -2*k/dx**2],
[-0.5*c/dx + k/dx**2]])
```

We compute the Jacobian in a sparse form. These object are callable, and will return the numerical values if we provide the fields and the parameters:

```
>>> print(model.F(fields, parameters))
array([...])

>>> print(model.J(fields, parameters))
<NxN sparse matrix of type '<class 'numpy.float64'>'
with M stored elements in Compressed Sparse Column format>
```

a numerical approximation is available for debug purpose with

```
>>> print(model.F(fields, parameters))
array([[...]])
```

be aware that numerical approximation of the Jacobian is far less efficient than the provided optimized routine.

2.2.1 optional arguments : fields and parameters

The model take two mandatory parameters: differential_equations, dependent_variables. The first define the evaluation of the time derivative, the second the name of the dependant variables.

It can take two optional arguments :

- parameters, a list of parameters name. They can be scalar or vector with the same dimension as the dependant variables.

- help_functions, a list of outside variables : they have to be vector with the same dimension of the dependant variable.

So, what is main difference between them? The difference is that you have the possibility to use spatial derivative of the fields in the model. Because the fields are parsed and the derivative approximated, it make the graph optimization of the model grows.

2.3 Model compilation

The model has to be compiled before being employed. The sympy library provides an easy way to automatically write the Fortran or C routine corresponding. Better than that, a tool has been written in order to convert sympy complex expressions to Theano graph which can be easily compiled.

In the examples folder live some classic 1D PDE (diffusion, diffusion/advection, burger equation...).

The Model class is pickable, means that it can be sent across the network and between cpu for multiprocessing purpose. It can be save on disk as a binary and reload later. It is important in order to reduce the large compilation overhead. (see Model.save and load_model). Thus, the model has to be re-optimized by Theano on every new host, leading to potential long initialization for large and complex models. The memory footprint can be large (> 1Go) in some case: this is the cost of the theano aggressive graph optimization strategy. [Further work will include the choice between high performance and fast overhead]. It should be important to notice that Theano is able to handle GPU computation if properly configured (see the Theano documentation for more details).

2.4 Fields containers

A special container has been designed to handle initial values of the dependant solutions (the unknowns), the independant variables (spatial coordinates), the constant fields and the post-processed variable (known as helper function).

A factory is linked to the model and is accessible via the model.fields_template property :

```
>>> import numpy as np
>>> from triflow import Model

>>> model = Model("k * dxxU - c * dxU",
...               "U", ["k", "c"])

>>> x, dx = np.linspace(0, 1, 100, retstep=True)
>>> U = np.cos(2 * np.pi * x * 5)
>>> fields = model.fields_template(x=x, U=U)
```

The variable involved in the computation are stored on a large vector containing all the fields, and this object give access to each fields to simplify their modification and the computations.

```
>>> fields.U[:] = 5
>>> print(fields.U)
[5, 5, 5, ..., 5, 5]
```

Be aware of difference between the attribute giving access to a view of the main array and the one returning a copy of the subarray: the first one allow an on-the-fly modification of the fields (in order to inject boundary condition for exemple), the second one should be only used as read-only meaning.

2.5 Numerical scheme, temporal solver

In order to provide fast and scalable temporal solver, the Jacobian use the scipy sparse column matrix format (which will reduce the memory usage, especialy for a large number of spatial nodes), and make available the SuperLU decomposition, a fast LU sparse matrix decomposition algorithm.

Different temporal schemes are provided in the plugins module:

- a forward Euler scheme
- a backward Euler scheme
- a θ mixed scheme
- A ROW schemes from order 3 up to 6 with fixed and variable time stepping.
- A proxy schemes giving access to all the scipy.integrate.ode schemes.

Each of them have advantages and disadvantages.

They can accept somme extra arguments during their instantiation (for exemple the θ parameter for the θ mixed scheme), and are called with the actual fields, time, time-step, parameters, and accept an optionnal hook modifying fields and parameters each time the solver compute the function or its jacobian.

The following code compute juste one time-step with a Crank-Nicolson scheme.

```
import numpy as np
import pylab as pl
from triflow import Model, schemes

pl.style.use('seaborn-whitegrid')

model = Model("k * dxxU - c * dxU",
              "U", ["k", "c"])

x, dx = np.linspace(0, 1, 200, retstep=True)
```

```
U = np.cos(2 * np.pi * x * 5)
fields = model.fields_template(x=x, U=U)

pl.plot(fields.x, fields.U)

parameters = dict(c=.03, k=.001, dx=dx, periodic=True)

t = 0
dt = 1E-1

scheme = schemes.Theta(model, theta=.5)   # Crank-Nicolson scheme

new_t, new_fields = scheme(t, fields, dt, parameters)

pl.plot(new_fields.x, new_fields.U)
pl.xlim(0, 1)
pl.show()
```
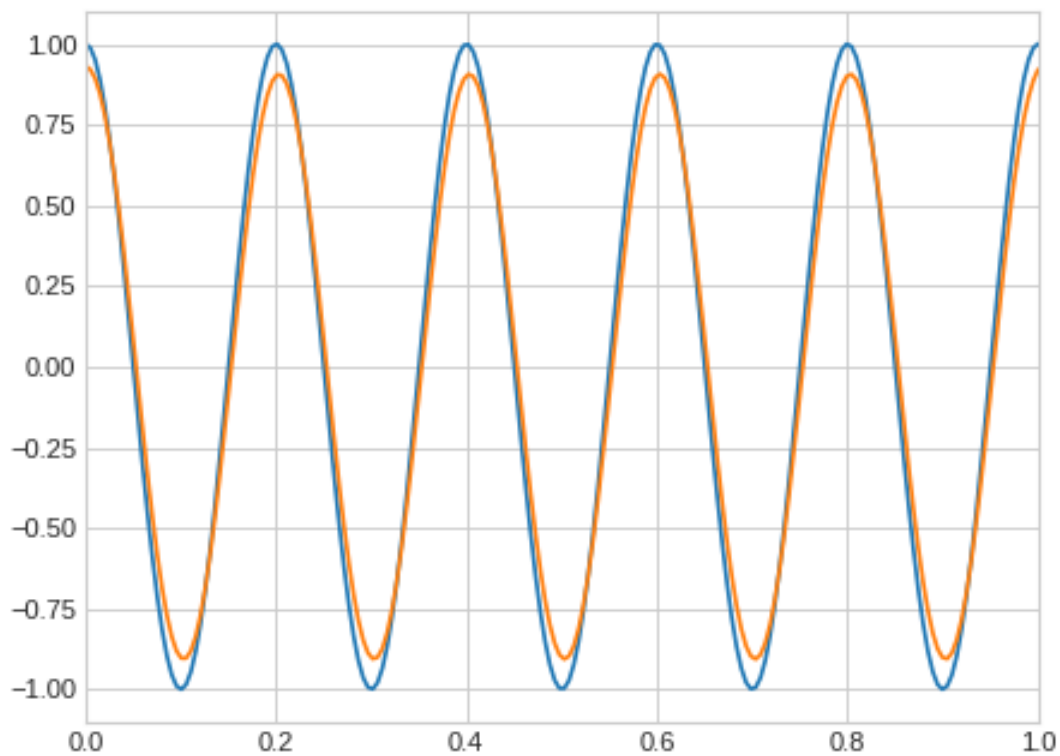

We obtain with the following code a full resolution up to the target time.

```
import itertools as it

import numpy as np
import pylab as pl
from triflow import Model, schemes
```

```python
pl.style.use('seaborn-whitegrid')

model = Model("k * dxxU - c * dxU",
              "U", ["k", "c"])

x, dx = np.linspace(0, 1, 200, retstep=True)
U = np.cos(2 * np.pi * x * 5)
fields = model.fields_template(x=x, U=U)

parameters = dict(c=.03, k=.001, dx=dx, periodic=True)

t = 0
dt = 1E-2
tmax = 5

pl.plot(fields.x, fields.U, label=f't: {t:g}')

scheme = schemes.Theta(model, theta=.5)  # Crank-Nicolson scheme

for i in it.count():
    t, fields = scheme(t, fields, dt, parameters)
    if (i + 1) % 100 == 0:
        pl.plot(fields.x, fields.U, label=f't: {t:g}')
    if t >= tmax:
        break

pl.xlim(0, 1)
legend = pl.legend(loc='best')

pl.show()
```
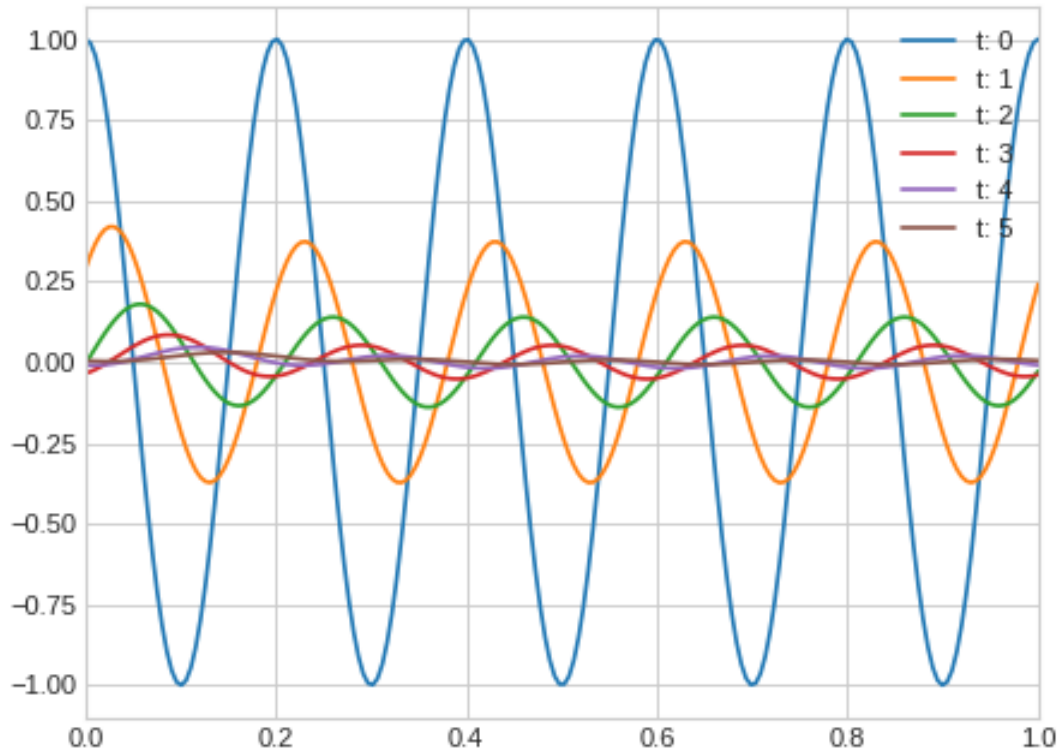
2.5.1 hook and boundary conditions

The hook function is used in order to deal with variable and conditional parameters and boundary condition.

Inside the model, the fields are padded in order to solve the equation. If the parameter "periodic" is used, the pad function is used with the mode "wrap" leading to periodic fields. If not, the mode "edge" is used, repeating the first and last node. It is very easy to implement Dirichlet condition with the following function:

```python
import itertools as it

import numpy as np
import pylab as pl
from triflow import Model, schemes

pl.style.use('seaborn-whitegrid')

model = Model("k * dxxU - c * dxU",
              "U", ["k", "c"])

x, dx = np.linspace(0, 1, 200, retstep=True)
U = np.cos(2 * np.pi * x * 5)
fields = model.fields_template(x=x, U=U)

parameters = dict(c=.03, k=.001, dx=dx, periodic=False)
```

triflow Documentation, Release 0.4.0

```python
t = 0
dt = 5E-1
tmax = 2.5

pl.plot(fields.x, fields.U, label=f't: {t:g}')

scheme = schemes.RODASPR(model)


def dirichlet_condition(t, fields, pars):
    fields.U[0] = 1
    fields.U[-1] = 0
    return fields, pars


for i in it.count():
    t, fields = scheme(t, fields, dt, parameters, hook=dirichlet_condition)
    print(f"iteration: {i}\t",
          f"t: {t:g}", end='\r')
    pl.plot(fields.x, fields.U, label=f't: {t:g}')
    if t >= tmax:
        break

pl.xlim(0, 1)
legend = pl.legend(loc='best')

pl.show()
```
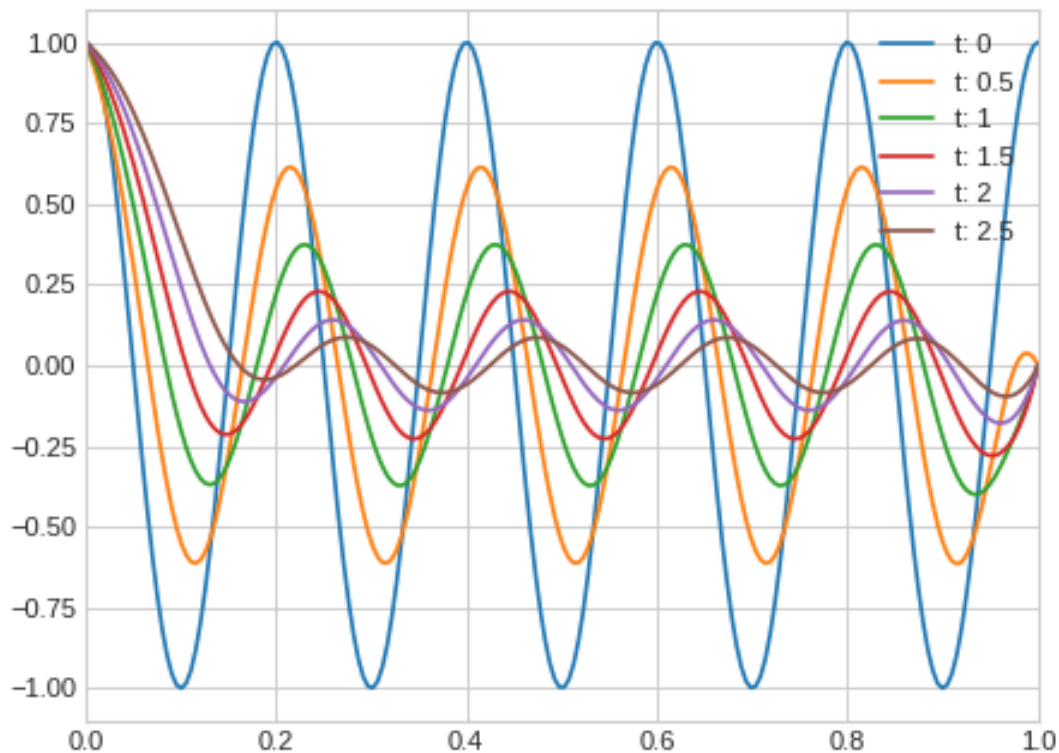
2.5. Numerical scheme, temporal solver   **9**

2.6 Simulation class: higher level control

The loop snippet

```
>>> scheme = schemes.RODASPR(model)
>>> for i in it.count():
...     t, fields = scheme(t, fields, dt, parameters)
...     print(f"iteration: {i}\tt: {t:g}", end='\r')
...     if t >= tmax:
...         break
```

is not handy.

To avoid it, we provide a higher level control class, the Simulation. It is an iterable and we can write the snippet as:

```
>>> simul = Simulation(model, t, fields, parameters, dt,
...                     scheme=schemes.RODASPR(model), tmax=tmax)
>>> for i, (t, fields) in enumerate(simul):
...     print(f"iteration: {i}\tt: {t:g}", end='\r')
```

and we write the previous advection-diffusion example as:

```
import numpy as np
import pylab as pl
```

```python
from triflow import Model, Simulation

pl.style.use('seaborn-whitegrid')

model = Model("k * dxxU - c * dxU",
              "U", ["k", "c"])

x, dx = np.linspace(0, 1, 200, retstep=True)
U = np.cos(2 * np.pi * x * 5)
fields = model.fields_template(x=x, U=U)

parameters = dict(c=.03, k=.001, dx=dx, periodic=False)

t = 0
dt = 5E-1
tmax = 2.5

pl.plot(fields.x, fields.U, label=f't: {t:g}')


def dirichlet_condition(t, fields, pars):
    fields.U[0] = 1
    fields.U[-1] = 0
    return fields, pars


simul = Simulation(model, t, fields, parameters, dt,
                   hook=dirichlet_condition, tmax=tmax)

for i, (t, fields) in enumerate(simul):
    print(f"iteration: {i}\t",
          f"t: {t:g}", end='\r')
    pl.plot(fields.x, fields.U, label=f't: {t:g}')

pl.xlim(0, 1)
legend = pl.legend(loc='best')

pl.show()
```
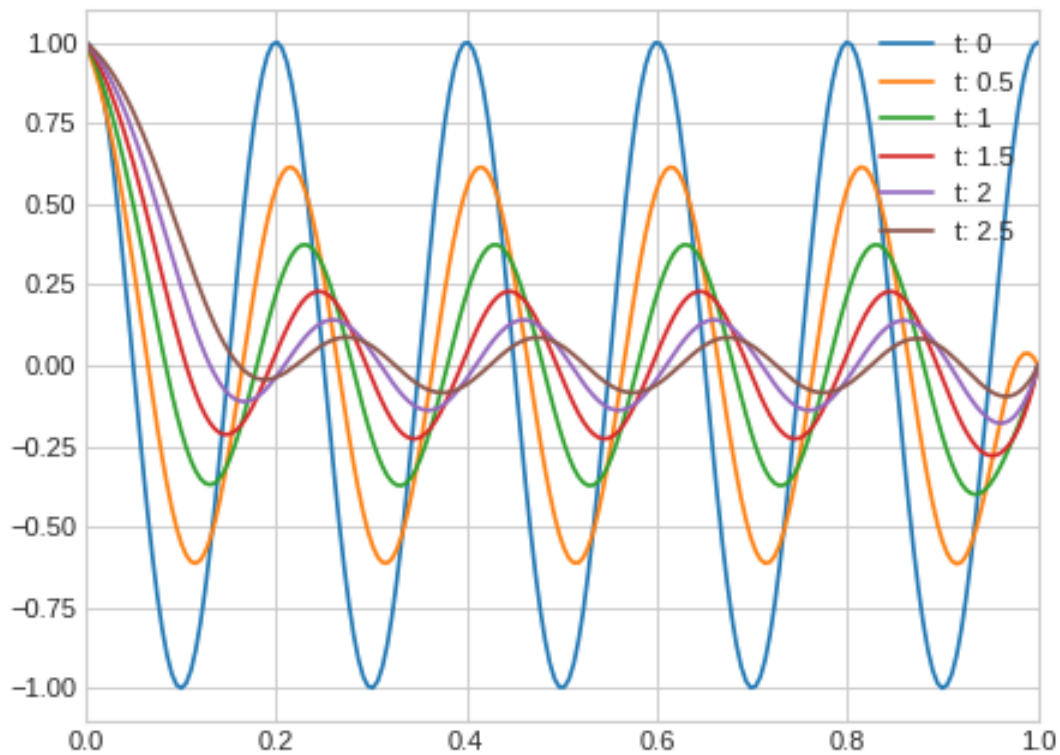
2.6.1 Displays

Hooks are called every internal time step and allow granular modification of the parameters or fields.

Displays have to be called by the user and can not modify the fields or parameters, but can display or save data during the simulation.

Like the hooks, they are basically callable or coroutine taking fields or the other to output post-processed data. The built-ins displays are detailed on the section of the same name.

TEMPORAL SCHEMES

This section will present the structure of a typical temporal scheme and how to write your own schemes.

3.1 List of available schemes

3.1.1 schemes.Theta

```
>>> scheme = schemes.Theta(model, theta)
```

This scheme represent a combinaison of the forward and the backward Euler. With theta = 0, it will be a full forward Euler, with theta = 1, a full backward Euler and with theta = 0.5, we will have a Crank-Nicolson method.

3.1.2 schemes.scipy_ode

```
>>> scheme = schemes.scipy_ode(model, integrator, **kwd_integrator)
```

This scheme is a wrapper around the scipy.integrate.ode.

The integrator is one of these provided by scipy and kwd_integrator allow us to pass extra parameters to the solver.

Beware that this scheme do not use the sparse jacobian, leading to higher memory usage and possibly bad performance for large systems. However, the time-stepping provided is good and is a good choice for validate a new scheme.

3.1.3 schemes.ROW_general

http://www.digibib.tu-bs.de/?docid=00055262 Rang, Joachim: Improved traditional Rosenbrock-Wanner methods for stiff odes and daes / Joachim Rang.

This is the parent of all the Rosenbrock-Wanner scheme provided: they follow the same algorithm with different number of internal steps and different coefficients. A time-stepping is available and these schemes are suitable for stiff equations.

- ROS2 (2 steps, only fixed time-step)

- ROS3PRw (3 steps)

- ROS3PRL (4 steps)

- RODASPR (6 steps)

3.2 Internal structure of a scheme

A temporal scheme can be written as any callable object initiated with a model attribute (which will give access to the system of differential equation to solve and its jacobian with model.F and model.J).

The *__call__* method have the following signature:

```
fields, t = scheme(fields, t, dt, pars,
                   hook=lambda fields, t, pars: (fields, pars))
```

It will take as input the actual fields container, the time and the time-step wanted for this step. As keyword argument it will take a hook, a callable with the fields, time and parameters as input and fields and parameters as output. This function give us the ability to make on-the-fly modification of the fields (for boundary condition), or parameters (allowing time and space conditional parameters).

This hook has to be called before calling the model routines.

```python
class BackwardEuler:

    def __init__(self, model):
        self.model = model

    def __call__(self, fields, t, dt, pars,
                 hook=lambda fields, t, pars: (fields, pars)):
        fields = fields.copy()
        fields, pars = hook(fields, t, pars)
        F = self.model.F(fields, pars)
        J = self.model.J(fields, pars)
        # access the flatten copy of the dependant variables
        U = fields.uflat
        B = dt * (F -  J @ U) + U
        J = (sps.identity(U.size,
                          format='csc') - dt * J)
        # used in order to update the value of the dependant variables
        fields = fields.fill(solver(J, B))
        # We return the hooked fields, be sure that the bdc are taken into account.
        fields, _ = hook(fields, t + dt, pars)
        return fields, t + dt
```

COOKBOOK

4.1 The convection diffusion equation

```python
import functools as ft
import multiprocessing as mp
import logging

import numpy as np
from scipy.signal import gaussian

import pylab as pl

from triflow import Model, Simulation, schemes, displays

pl.style.use('seaborn-white')

%matplotlib inline
```

The convection–diffusion equation is a combination of the diffusion and convection (advection) equations, and describes physical phenomena where particles, energy, or other physical quantities are transferred inside a physical system due to two processes: diffusion and convection. (Wikipedia)

The equation reads

$$\partial_t U = k\partial_{xx}U - c\partial_x U$$

with

- U the physical quantities transferred (it could be a chemical species concentration, the temperature of a fluid...)

- k a diffusion convection

- c a velocity, which will be constant in our example.

```python
model = Model("k * dxxU - c * dxU",
              "U", ["k", "c"])
```

We discretize our spatial domain. `retstep=True` ask to return the spatial step. We want periodic condition, so `endpoint=True` exclude the final node (which will be redondant with the first node, $x = 0$ and $x = 100$ are merged)
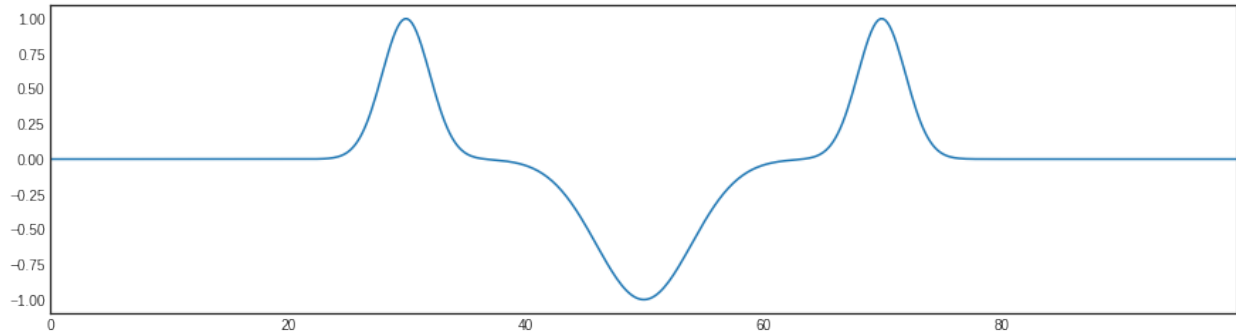
```python
x, dx = np.linspace(0, 100, 500, retstep=True, endpoint=False)
```

We initialize with three gaussian pulses for the initial condition

```
U = (np.roll(gaussian(x.size, 10), x.size // 5) +
     np.roll(gaussian(x.size, 10), -x.size // 5) -
     gaussian(x.size, 20))

fields = model.fields_template(x=x, U=U)

pl.figure(figsize=(15, 4))
pl.plot(fields.x, fields.U)
pl.xlim(0, fields.x.max())
pl.show()
```


We precise our parameters. The default scheme provide an automatic time_stepping. We set the periodic flag to True.

```
parameters = dict(k=.2, c=10, periodic=True)
```

We initialize the simulation.

```
t = 0
simulation = Simulation(model, t, fields, parameters,
                        dt=.1, tmax=30)
```

We iterate on the simulation until the end. We update the bokeh display each step.

```
pl.figure(figsize=(15, 10))
for i, (t, fields) in enumerate(simulation):
    if i % 2 == 0:
        pl.fill_between(fields.x, fields.U + .1 * (i + 1),
                        fields.U.min() - 1,
                        color='darkred', zorder=-2 * i, alpha=.7)
        pl.plot(fields.x, fields.U + .1 * (i + 1),
                color='white',
                zorder=-(2 * i - 1))
    print(f"t: {t:g}".ljust(80), end='\r')
pl.xlim(0, fields.x.max())
pl.show()
```

```
t: 29.9
```
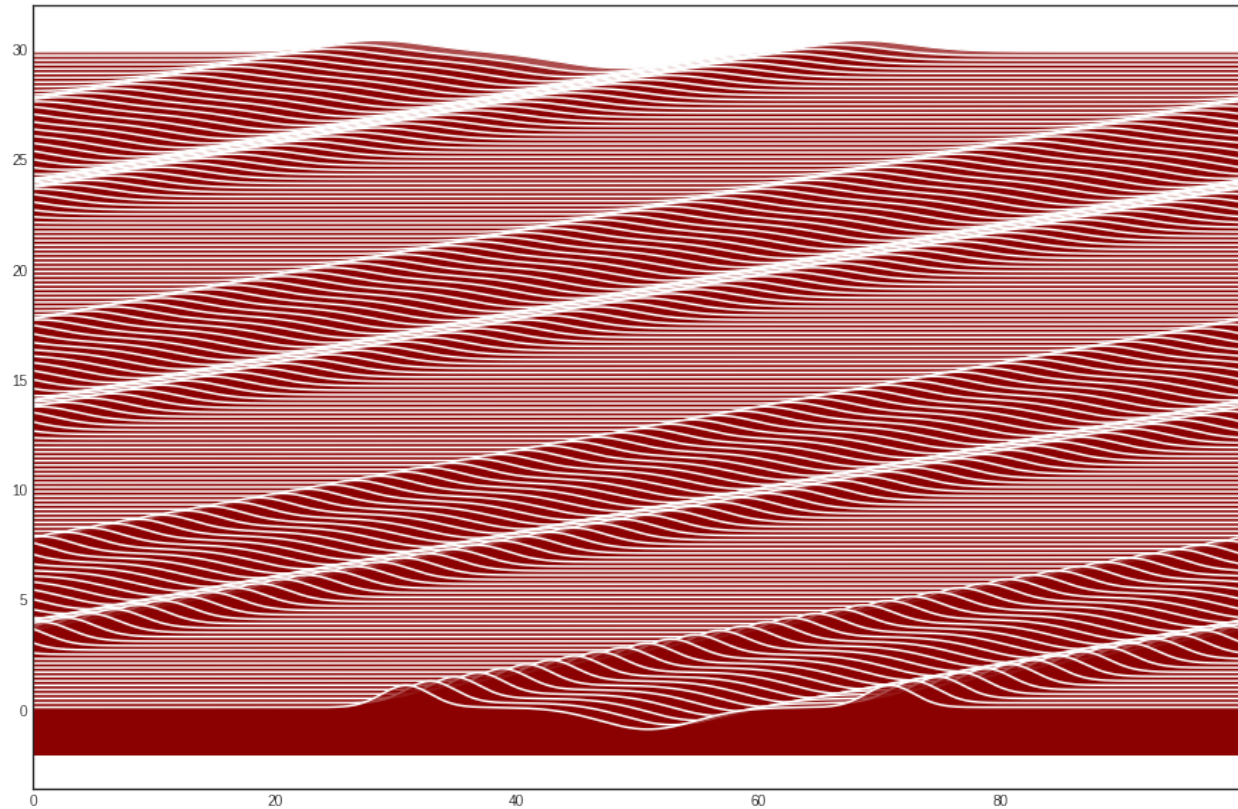
4.2 The burger equation

```python
import functools as ft
import multiprocessing as mp
import logging

import numpy as np
from scipy.signal import gaussian

import pylab as pl

from triflow import Model, Simulation, schemes, displays

pl.style.use('seaborn-white')

%matplotlib inline
```

Burgers' equation is a fundamental partial differential equation occurring in various areas of applied mathematics, such as fluid mechanics, nonlinear acoustics, gas dynamics, traffic flow. It is named for Johannes Martinus Burgers (1895–1981). (Wikipedia)

The viscous Burger equation in 1D reads:

$$\partial_t U = k\partial_{xx}U - U\partial_x U$$

with

- U the velocity

- k a diffusion convection

The expected behaviour is a wave moving forward with a growing shock. This shock leads to discontinuity smoothed by the diffusion term.

```
model = Model("k * dxxU - U * dxU",
              "U", "k")
```

We discretize our spatial domain. `retstep=True` ask to return the spatial step. We want periodic condition, so `endpoint=True` exclude the final node (which will be redondant with the first node, $x = 0$ and $x = 100$ are merged)
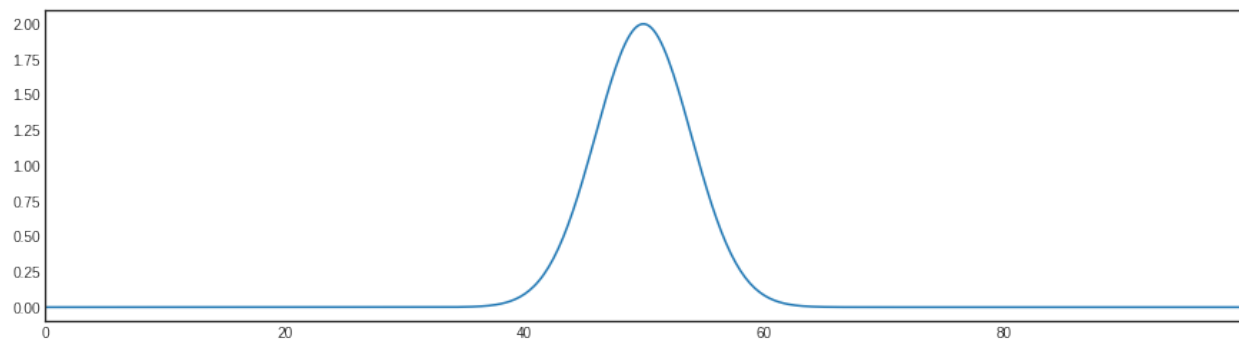
```
x, dx = np.linspace(0, 100, 500, retstep=True, endpoint=False)
```

We initialize with a simple gaussian pulse initial condition.

```
U = gaussian(x.size, 20) * 2

fields = model.fields_template(x=x, U=U)

pl.figure(figsize=(15, 4))
pl.plot(fields.x, fields.U)
pl.xlim(0, fields.x.max())
pl.show()
```


We precise our parameters. The default scheme provide an automatic time_stepping. We set the periodic flag to True

```
parameters = dict(k=1E-1, c=20, periodic=True)
```

We initialize the simulation.

```
t = 0
simulation = Simulation(model, t, fields, parameters,
                        dt=.1, tmax=5, tol=1E-4)
```

We iterate on the simulation until the end.

```
pl.figure(figsize=(15, 10))
for i, (t, fields) in enumerate(simulation):
    if i % 2 == 0:
        pl.fill_between(fields.x, fields.U + .1 * (i + 1),
                        fields.U.min() - 1,
                        color='darkred', zorder=-2 * i, alpha=.7)
        pl.plot(fields.x, fields.U + .1 * (i + 1),
                color='white',
                zorder=-(2 * i) + 1)
```

```
    print(f"t: {t:g}".ljust(80), end='\r')
pl.xlim(0, fields.x.max())
pl.show()
```

```
t: 5
```
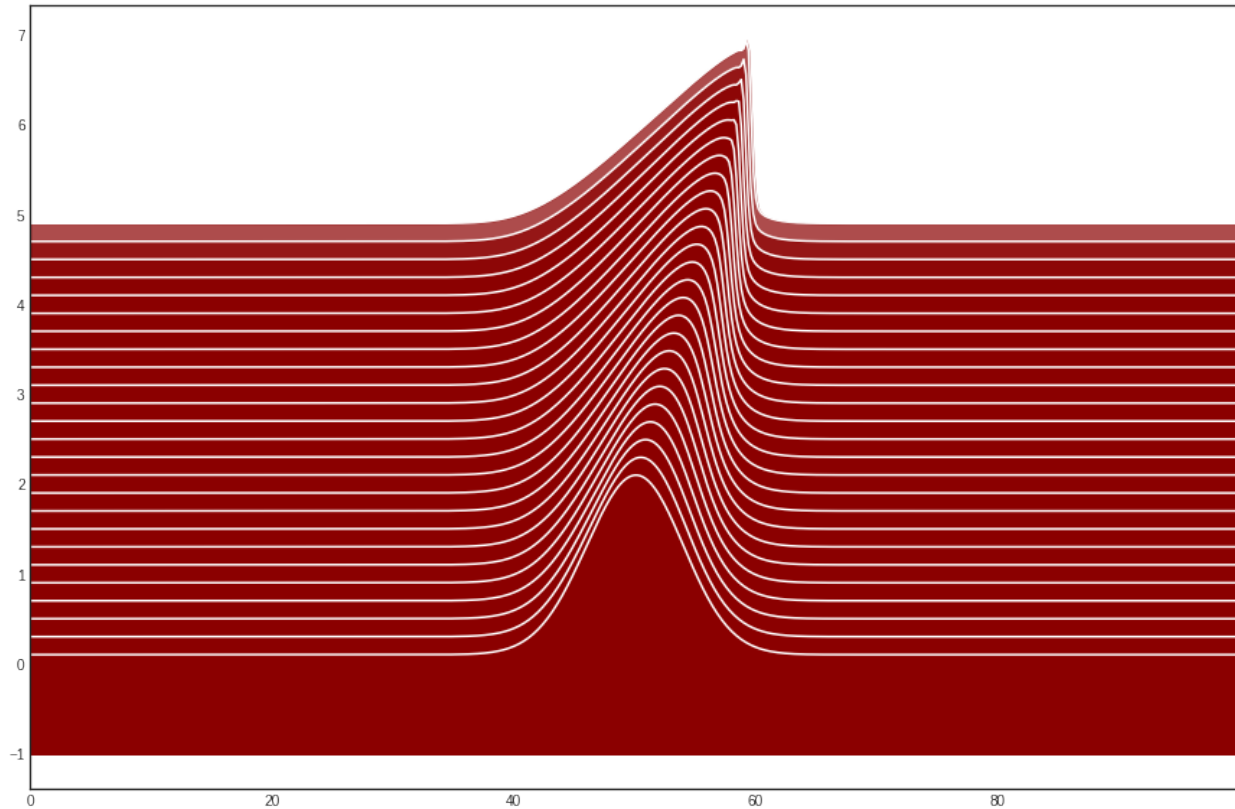

4.3 Wave equation

```python
import functools as ft
import multiprocessing as mp
import logging

import numpy as np
from scipy.signal import gaussian

import pylab as pl

from triflow import Model, Simulation, schemes, displays

%matplotlib inline
```

We initialize the model with the wave equation written as a system of first order differential equations.

$$\partial_{t,\,t}u = c^2\partial_{x,\,x}u$$

which lead to

$$\partial_t u = v \tag{4.1}$$

$$\partial_t v = c^2 \partial_{x,\,x} u \tag{4.2}$$

with c the velocity of the wave.

```
model = Model(["c**2 * dxxu", "v"],
              ["v", "u"], "c")
```

We discretize our spatial domain. `retstep=True` ask to return the spatial step.
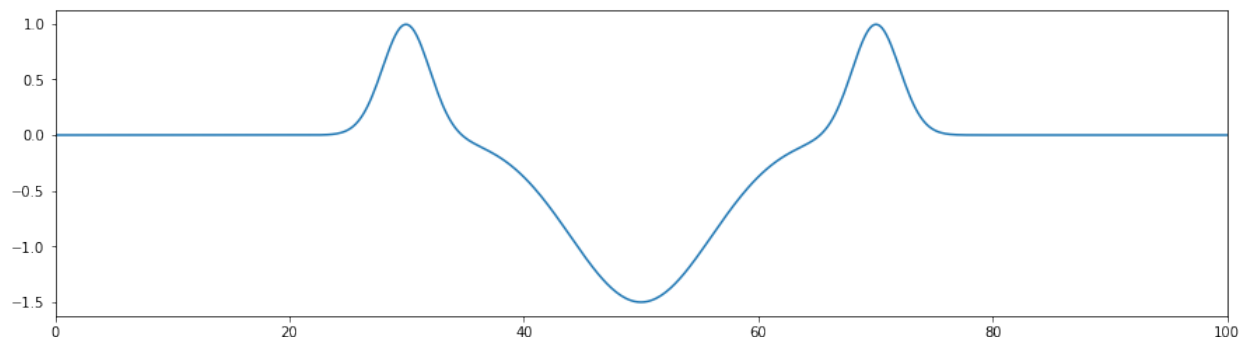
```
x, dx = np.linspace(0, 100, 500, retstep=True)
```

We initialize with three gaussian pulses for the initial condition

```
u = (np.roll(gaussian(x.size, 10), x.size // 5) +
     np.roll(gaussian(x.size, 10), -x.size // 5) -
     gaussian(x.size, 30) * 1.5)
v = np.zeros_like(u)

fields = model.fields_template(x=x, u=u, v=v)

pl.figure(figsize=(15, 4))
pl.plot(fields.x, fields.u)
pl.xlim(0, fields.x.max())
pl.show()
```


We precise our parameters. The default scheme provide an automatic time_stepping. We want dirichlet boundary condition, so we set the periodic flag to False.

```
parameters = dict(c=5, periodic=False)
```

This function will set the boundary condition. We will have a fixed rope at each edge.

```
def dirichlet(t, fields, pars):
    # fields.u[:] = np.sin(t * 2 * np.pi * 2) * gaussian(x.size, 10) - fields.u[:]
    fields.u[0] = 0
    fields.u[-1] = 0
    fields.v[0] = 0
    fields.v[-1] = 0
    return fields, pars
```

We initialize the simulation, and we set a bokeh display in order to have real-time plotting.

```
t = 0
simulation = Simulation(model, t, fields, parameters,
                        dt=.1, tmax=15,
                        hook=dirichlet)
```

We iterate on the simulation until the end. We update the bokeh display each step.

```
pl.figure(figsize=(15, 10))
for i, (t, fields) in enumerate(simulation):
    if i % 2 == 0:
        pl.fill_between(fields.x, fields.u + .1 * (i + 1),
                        fields.u.min() - 1,
                        color='darkred', zorder=-2 * i, alpha=.7)
        pl.plot(fields.x, fields.u + .1 * (i + 1),
                color='white',
                zorder=-(2 * i - 1))
    print(f"t: {t:g}".ljust(80), end='\r')
pl.xlim(0, fields.x.max())
pl.show()
```

```
t: 15
```
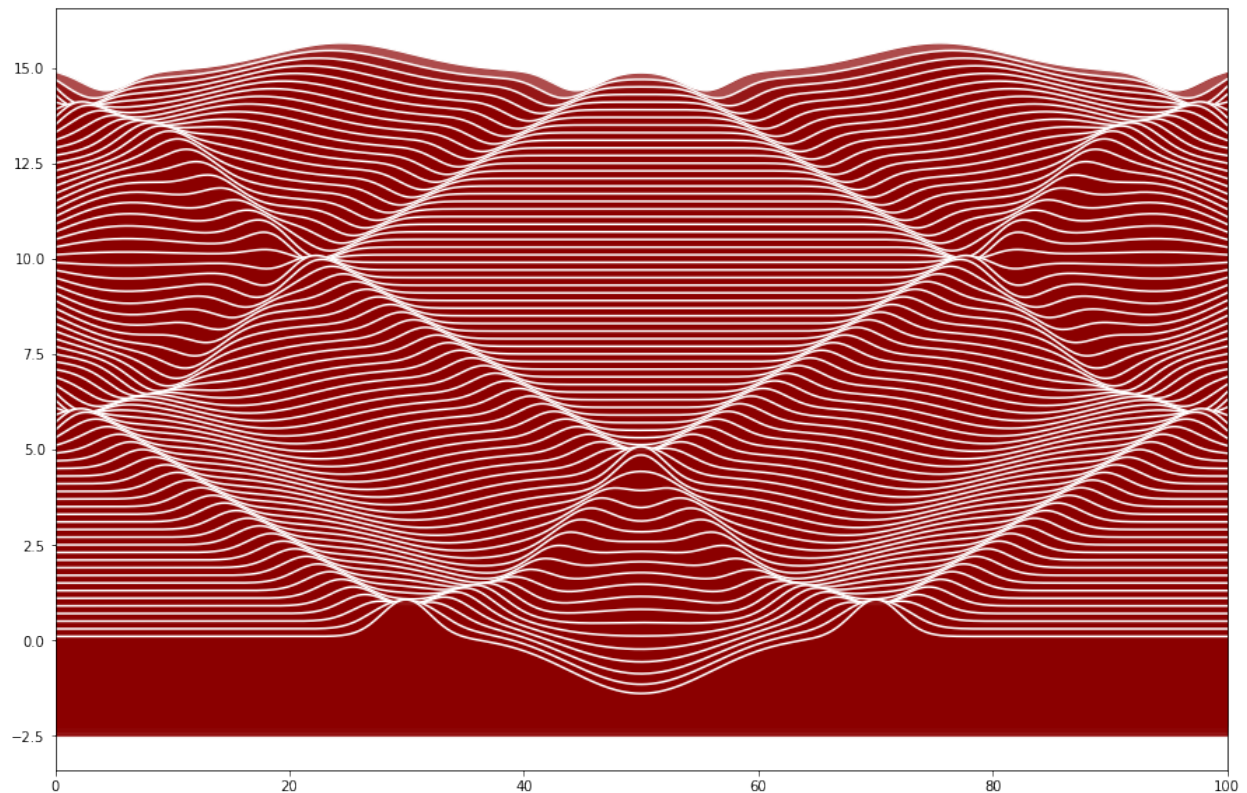

4.4 Coupled burger's-like equations

```
import functools as ft
import multiprocessing as mp
```

```
import logging

import numpy as np
from scipy.signal import gaussian

import pylab as pl

from triflow import Model, Simulation, schemes, displays

%matplotlib inline
```

We initialize the model with a coupled burger-like system of equations

$$\partial_t U = k\partial_{xx}U - cU\partial_x V \tag{4.3}$$
$$\partial_t V = k\partial_{xx}U - cV\partial_x U \tag{4.4}$$

```
model = Model(["k * dxxU - c * U * dxV", "k * dxxV - c * V * dxU"],
              ["U", "V"], ["k", "c"])
```

We discretize our spatial domain. `retstep=True` ask to return the spatial step. We want periodic condition, so `endpoint=True` exclude the final node (which will be redondant with the first node, $x = 0$ and $x = 100$ are merged)

```
x, dx = np.linspace(0, 100, 500, retstep=True, endpoint=False)
```

We initialize with cosine and sine function for U and V.

```
U = np.cos(x * 2 * np.pi / x.max() * 5) * .5 + 1
V = np.sin(x * 2 * np.pi / x.max() * 5) * .5 + 1
fields = model.fields_template(x=x, U=U, V=V)
```

We precise our parameters. The default scheme provide an automatic time_stepping. We set the periodic flag to True.

```
parameters = dict(k=1, c=10, periodic=True)
```

We initialize the simulation

```
t = 0
simulation = Simulation(model, t, fields, parameters,
                        dt=.1, tmax=4, tol=1E-2)
```

We iterate on the simulation until the end. We update the bokeh display each step.

```
pl.figure(figsize=(20, 12))
for i, (t, fields) in enumerate(simulation):
    if i % 2 == 0:
        pl.fill_between(fields.x, fields.U + .1 * (2 * i),
                        fields.U.min() - 1,
                        color='darkred', zorder=-4 * i, alpha=.7)
        pl.plot(fields.x, fields.U + .1 * (2 * i),
                color='white',
                zorder=-4 * i + 1)
        pl.fill_between(fields.x, fields.V + .1 * (2 * i + 1),
                        fields.V.min() - 1,
                        color='lightsteelblue', zorder=-4 * i + 2, alpha=.7)
        pl.plot(fields.x, fields.V + .1 * (2 * i + 1),
```

```
                color='white',
                zorder=-4 * i + 3)
    print(f"t: {t:g}".ljust(80), end='\r')
pl.xlim(0, fields.x.max())
pl.show()
```

```
t: 3.9
```

FIVE
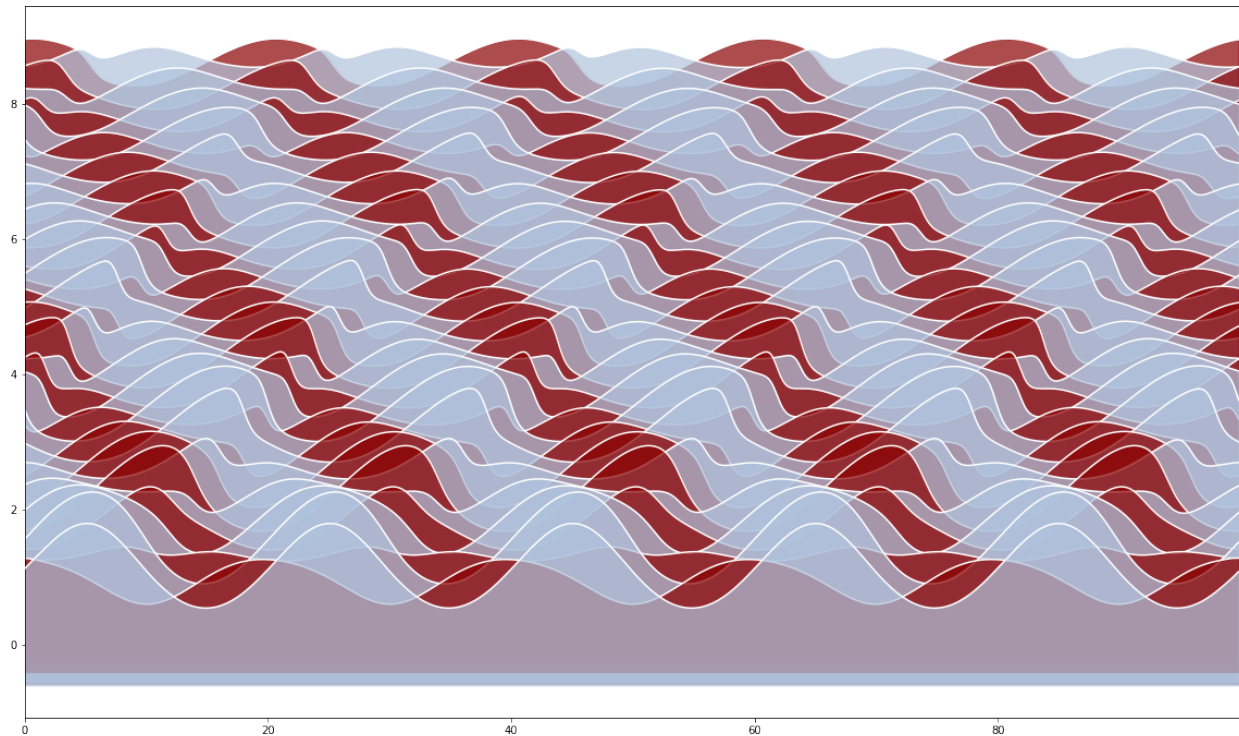
TRIFLOW

5.1 triflow package

5.1.1 Subpackages

5.1.1.1 triflow.core package

Submodules

triflow.core.fields module

class `triflow.core.fields.`**`BaseFields`**(***inputs*)

    Bases: `object`

    Specialized container which expose the data as a structured numpy array, give access to the dependants variables and the herlpers function as attributes (as a numpy rec array) and is able to give access to a flat view of the dependent variables only (which is needed by the ode solvers for all the linear algebra manipulation).

        Parameters ****inputs** (*numpy.array*) – named argument providing x, the dependent variables and the helper functions. All of these are mendatory and a KeyError will be raised if a data is missing.

    array

        numpy.array

    vanilla numpy array containing the data

    size

        int

    Number of discretisation nodes

    copy()

    static **factory**(*dependent_variables*, *helper_functions*)

        Fields factory generating specialized container build around a triflow Model.

        Parameters

            • **dependent_variables** (*iterable of str*) – name of the dependent variables

            • **helper_functions** (*iterable of str*) – name of the helper functions

        Returns Specialized container which expose the data as a structured numpy array

        Return type triflow.BaseFields

fill(*Uflat*)

take a flat numpy array and update inplace the dependent variables of the container

> **Returns**
>
> **Return type** None
>
> **Parameters Uflat** (*TYPE*) – Description

flat

return a flat view of the fields

Returns: numpy.ndarray.view: flat view of the main numpy array

structured

return a structured array of the main numpy array as a view

Returns: numpy.ndarray.view: structured view of the main numpy array

uarray

return a view array of the main numpy array with only the dependant variables.

Returns: numpy.ndarray.view: view of the dependent variables of the main numpy array

uflat

return a flatten **copy** of the main numpy array with only the dependant variables.

Be carefull, modification of these data will not be reflected on the main array!

Returns: numpy.ndarray: **copy** of the dependent variables of the main numpy array

triflow.core.model module

class triflow.core.model.**Model**(*differential_equations*, *dependent_variables*, *parameters=None*, *help_functions=None*)

Bases: object

Contain finite difference approximation and routine of the dynamical system

> Take a mathematical form as input, use Sympy to transform it as a symbolic expression, perform the finite difference approximation and expose theano optimized routine for both the right hand side of the dynamical system and Jacobian matrix approximation.
>
> **differential_equations** [iterable of str or str] the right hand sides of the partial differential equations written as :math:`

rac{partial U}{partial t} = F(U)`, where the spatial derivative can be written as *dxxU* or *dx(U, 2)* or with the sympy notatio

> **dependent_variables** [iterable of str or str] the dependent variables with the same order as the temporal derivative of the previous arg.
>
> **parameters** [iterable of str or str, optional, default None] list of the parameters. Can be feed with a scalar of an array with the same size
>
> **help_functions** [None, optional] All fields which have not to be solved with the time derivative but will be derived in space.
>
> F : triflow.F_Routine Callable used to compute the right hand side of the dynamical system F_array : numpy.ndarray of sympy.Expr Symbolic expressions of the right hand side of the dynamical system J : triflow.J_Routine Callable used to compute the Jacobian of the dynamical system J_array : numpy.ndarray of sympy.Expr Symbolic expressions of the Jacobian side of the dynamical system

fields_template: Model specific Fields container used to store and access to the model variables in an efficient way.

save: Save a binary of the Model with pre-optimized F and J routines

A simple diffusion equation:

```
>>> from triflow import Model
>>> model = Model("k * dxxU", "U", "k")
```

A coupled system of convection-diffusion equation:

```
>>> from triflow import Model
>>> model = Model(["k1 * dxxU - c1 * dxV",
...                "k2 * dxxV - c2 * dxU",],
...               ["U", "V"], ["k1", "k2", "c1", "c2"])
```

Include t as symbolic variable to allow time dependent functions (non-autonomous dynamical systems)

fields_template

sargs

save (*filename*)
Save the model as a binary pickle file.

> **Parameters** **filename** (*str*) – name of the file where the model is saved.
>
> **Returns**
>
> **Return type** None

triflow.core.model.**load_model** (*filename*)
load a pre-compiled triflow model. The internal of theano allow a caching of the model. Will be slow if it is the first time the model is loaded on the system.

> **Parameters** **filename** (*str*) – path of the pre-compiled model
>
> **Returns** triflow pre-compiled model
>
> **Return type** triflow.core.Model

triflow.core.routines module

class triflow.core.routines.**F_Routine** (*matrix*, *args*, *pars*, *ufunc*, *reduced=False*)
Bases: *triflow.core.routines.ModelRoutine*

Compute the right hand side of the dynamical system :math:'

rac{partial U}{partial t} = F(U)'

> **fields** [triflow.Fields] triflow fields container generated by a triflow.Model containing the actual state of the dependent variables and helper functions.
>
> **pars** [dict] dictionnary with the different physical parameters of the model and the 'periodic' key.
>
> **numpy.ndarray** flat array containing the right hand side of the dynamical system.

diff_approx (*fields*, *pars*, *eps=1e-08*)

class triflow.core.routines.**J_Routine** (*matrix*, *args*, *pars*, *ufunc*, *reduced=False*)
Bases: *triflow.core.routines.ModelRoutine*

Compute the right hand side of the dynamical system :math:'

rac{partial U}{partial t} = F(U)'

> **fields** [triflow.Fields] triflow fields container generated by a triflow.Model containing the actual state of the dependent variables and helper functions.

> **pars** [dict] dictionnary with the different physical parameters of the model and the 'periodic' key.

> **sparse** [bool, optional, default True] whether should the matrix returned as dense or sparse form.

> scipy.sparse.CSC or numpy.ndarray: sparse or dense form (depending of the *sparse* argument) of the Jacobian approximation of the dynamical system right hand side.

class `triflow.core.routines.`**`ModelRoutine`** (*matrix*, *args*, *pars*, *ufunc*, *reduced=False*)

> Bases: `object`

triflow.core.simulation module

class `triflow.core.simulation.`**`Simulation`** (*model*, *t*, *fields*, *physical_parameters*, *dt*, *id=None*, *hook=<function Simulation.<lambda>>*, *scheme=<class 'triflow.plugins.schemes.RODASPR'>*, *tmax=None*, ***kwargs*)

> Bases: `object`

> High level container used to run simulation build on triflow Model. This object is an iterable which will yield every time step until the parameters 'tmax' is reached if provided. By default, the solver use a 6th order ROW solver, an implicit method with integrated time-stepping.

> > **Parameters**

> > - **model** (`triflow.Model`) – Contain finite difference approximation and routine of the dynamical system

> > - **t** (`float`) – initial time

> > - **fields** (`triflow.Fields`) – triflow container filled with initial conditions

> > - **physical_parameters** (`dict`) – physical parameters of the simulation

> > - **id** (`None, optional`) – name of the simulation. A 2 word slug will be generated if not provided.

> > - **hook** (`callable, optional`) – any callable taking the actual time, fields and parameters and return modified fields and parameters. Will be called every internal time step and can be used to include time dependent or conditionnal parameters, boundary conditions...

> > - **scheme** (`callable, optional, default triflow.schemes.RODASPR`) – an callable object which take the simulation state and return the next step. Its signature is scheme.__call__(fields, t, dt, pars, hook) and it should return the next time and the updated fields. It take the model and extra positional and named arguments.

> > - ****kwargs** (`*args,`) – extra arguments passed to the scheme.

> > - ****kwargs** – extra arguments passed to the scheme.

> **dt**
> > *float* – output time step

> **fields**
> > *triflow.Fields* – triflow container filled with actual data

i

> *int* – actual iteration

id

> *str* – name of the simulation

model

> *triflow.Model* – triflow Model used in the simulation

physical_parameters

> *dict* – physical parameters of the simulation

status

> *str* – status of the simulation, one of the following one: ('created', 'running', 'finished', 'failed')

t

> *float* – actual time

tmax

> *float or None, default None* – stopping time of the simulation. Not stopping if set to None.

Examples

```python
>>> import numpy as np
>>> import triflow
>>> model = triflow.Model(["k1 * dxxU",
...                         "k2 * dxxV"],
...                        ["U", "V"],
...                        ["k1", "k2"])
>>> x = np.linspace(0, 100, 1000, endpoint=False)
>>> U = np.cos(x * 2 * np.pi / 100)
>>> V = np.sin(x * 2 * np.pi / 100)
>>> fields = model.fields_template(x=x, U=U, V=V)
>>> pars = {'k1': 1, 'k2': 1, 'periodic': True}
>>> simulation = triflow.Simulation(model, 0, fields,
...                                 pars, dt=5, tmax=50)
>>> for t, fields in simulation:
...     pass
>>> print(t)
50
```

compute()

> Generator which yield the actual state of the system every dt.
>
> > **Yields** **tuple** (*t, fields*) – Actual time and updated fields container.

Module contents

5.1.1.2 triflow.plugins package

Submodules

triflow.plugins.displays module

This module regroups different displays: function and coroutine written in order to give extra informationto the user during the simulation (plot, post-processing...)

`triflow.plugins.displays.`**`amnesic_mean`**`()`
 Coroutine able to compute mean of a sample without keeping data in memory.

Examples

In that example we check if the mean value returned by the coroutine feeded by normal pseudo random values
goes close to 0

```
>>> import numpy as np
>>> from triflow.plugins import displays
>>> mean_coroutine = displays.amnesic_mean()
>>> for value in np.random.randn(5000):
...     x_mean = mean_coroutine.send(value)
```

`triflow.plugins.displays.`**`bokeh_fields_update`**`(keys=None,` *line_kwargs={},*
 fig_kwargs={})
 Coroutine sending fields data in a interactive Bokeh plot displayed in a jupyter notebook.
 bokeh.io.output_notebook() have to be called before using this coroutine.

> **Parameters**
>
> - **keys** (`None, optional`) – Description
> - **line_kwargs** (`dict of dict`) – dictionnary with vars as key and a dictionnary of keywords arguments passed to the lines plots
> - **fig_kwargs** (`dict of dict`) – dictionnary with vars as key and a dictionnary of keywords arguments passed to the figs plots

Examples

```
>>> from triflow import Model, Simulation, displays
>>> import numpy as np
```

```
>>> model = Model("dxxU", "U")
>>> parameters = dict(periodic=True)
```

```
>>> x = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=False)
>>> U = np.cos(x)
```

```
>>> fields = model.fields_template(x=x, U=U)
>>> simul = Simulation(model, 0, fields, parameters, dt=.5, tmax=10)
```

```
>>> bokeh_upt = displays.bokeh_fields_update()
```

```
>>> for t, fields in simul:
...     bokeh_upt.send((t, fields))
```

vars [list or None (default None)] list of the plotted variables. Default all the dependant variables of the model.

`triflow.plugins.displays.`**`bokeh_probes_update`**`(`*probes*, *line_kwargs={}*, *fig_kwargs={}*`)`
 Coroutine sending custom probes in a interactive Bokeh plot displayed in a jupyter notebook.
 bokeh.io.output_notebook() have to be called before using this coroutine.

Parameters

- **probes** (`dictionnary of callable`) – Dictionnary with {name: callable} used to plot the probes. The signature is the same as in the hooks and return the value we want to plot.

- **line_kwargs** (`dict of dict`) – dictionnary with vars as key and a dictionnary of keywords arguments passed to the lines plots

- **fig_kwargs** (`dict of dict`) – dictionnary with vars as key and a dictionnary of keywords arguments passed to the figs plots

Examples

```
>>> from triflow import Model, Simulation, displays
>>> import numpy as np
```

```
>>> model = Model("dxxU", "U")
>>> parameters = dict(periodic=True)
```

```
>>> x = np.linspace(-2 * np.pi, 2 * np.pi, 500, endpoint=False)
>>> U = np.cos(x) * 5
```

```
>>> fields = model.fields_template(x=x, U=U)
>>> simul = Simulation(model, 0, fields, parameters, dt=.01, tmax=1)
>>> def mean_probe(t, fields):
...     return np.mean(fields.U)
>>> bokeh_upt = displays.bokeh_probes_update({"mean": mean_probe})
```

```
>>> for t, fields in simul:
...     bokeh_upt.send((t, fields))
```

triflow.plugins.displays.**coroutine**(*func*)

triflow.plugins.displays.**visdom_update**(*addr='http://127.0.0.1'*, *env='main'*, *keys=None*)
Coroutine sending fields data in a interactive plot leaving on a visdom server (which has to be launched).

Parameters

- **addr** (str (default "http://127.0.0.1")) – adress of the visdom server.

- **env** (`str (default "main")`) – Visdom environnement where the plot will be send.

- **keys** (`None, optional`) – Description

Examples

```
>>> from triflow import Model, Simulation, displays
>>> import numpy as np
```

```
>>> model = Model("dxxU", "U")
>>> parameters = dict(periodic=True)
```

```
>>> x = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=False)
>>> U = np.cos(x)
```

```
>>> fields = model.fields_template(x=x, U=U)
>>> simul = Simulation(model, 0, fields, parameters, dt=.5, tmax=10)
>>> visdom_upt = displays.visdom_update()
>>> for t, fields in simul:
...     visdom_upt.send((t, fields))
```

> **vars** [list or None (default None)] list of the plotted variables. Default all the dependant variables of the model.

triflow.plugins.displays.**window_data**(*window_len=None*)

> Coroutine able to return a dictionnary filled with concatenated values of the time and the fields, with optionnal window lenght. Useful to save only a part of the solution.
>
> > **Parameters window_len** (*int or None (default)*) – number of last iterations kept. All data kept if None

Examples

```
>>> from triflow import Model, Simulation, displays
>>> import numpy as np
```

```
>>> model = Model("dxxU", "U")
>>> parameters = dict(periodic=True)
```

```
>>> x = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=False)
>>> U = np.cos(x)
```

```
>>> fields = model.fields_template(x=x, U=U)
>>> simul = Simulation(model, 0, fields, parameters, dt=.5, tmax=10)
```

```
>>> window_save_gen = displays.window_data()
```

```
>>> for t, fields in simul:
...     saved_data = window_save_gen.send((t, fields))
```

triflow.plugins.schemes module

This module regroups all the implemented temporal schemes. They are written as callable class which take the model and some control arguments at the init, and perform a computation step every time they are called.

The following solvers are implemented:

> - Backward and Forward Euler, Crank-Nicolson method (with the Theta class)
> - Some Rosenbrock Wanner schemes (up to the 6th order) with time controler
> - All the scipy.integrate.ode integrators with the scipy_ode class.

class triflow.plugins.schemes.**RODASPR**(*model*, *tol=0.01*, *time_stepping=True*, *max_iter=None*, *dt_min=None*)

> Bases: *triflow.plugins.schemes.ROW_general*
>
> 6th order Rosenbrock scheme, with time stepping
>
> > **Parameters**

- **model** (*triflow.Model*) – triflow Model

- **tol** (*float, optional, default 1E-2*) – tolerance factor for the time stepping. The time step will adapt to ensure that the maximum relative error on all fields stay under that value.

- **time_stepping** (*bool, optional, default True*) – allow a variable internal time-step to ensure good agreement between computing performance and accuracy.

- **max_iter** (*float or None, optional, default None*) – maximum internal iteration allowed

- **dt_min** (*float or None, optional, default None*) – minimum internal time step allowed

class triflow.plugins.schemes.**ROS2**(*model*)

Bases: *triflow.plugins.schemes.ROW_general*

Second order Rosenbrock scheme, without time stepping

> **Parameters model** (*triflow.Model*) – triflow Model

class triflow.plugins.schemes.**ROS3PRL**(*model*, *tol=0.01*, *time_stepping=True*, *max_iter=None*, *dt_min=None*)

Bases: *triflow.plugins.schemes.ROW_general*

4th order Rosenbrock scheme, with time stepping

Parameters

- **model** (*triflow.Model*) – triflow Model

- **tol** (*float, optional, default 1E-2*) – tolerance factor for the time stepping. The time step will adapt to ensure that the maximum relative error on all fields stay under that value.

- **time_stepping** (*bool, optional, default True*) – allow a variable internal time-step to ensure good agreement between computing performance and accuracy.

- **max_iter** (*float or None, optional, default None*) – maximum internal iteration allowed

- **dt_min** (*float or None, optional, default None*) – minimum internal time step allowed

class triflow.plugins.schemes.**ROS3PRw**(*model*, *tol=0.01*, *time_stepping=True*, *max_iter=None*, *dt_min=None*)

Bases: *triflow.plugins.schemes.ROW_general*

Third order Rosenbrock scheme, with time stepping

Parameters

- **model** (*triflow.Model*) – triflow Model

- **tol** (*float, optional, default 1E-2*) – tolerance factor for the time stepping. The time step will adapt to ensure that the maximum relative error on all fields stay under that value.

- **time_stepping** (*bool, optional, default True*) – allow a variable internal time-step to ensure good agreement between computing performance and accuracy.

- **max_iter** (*float or None, optional, default None*) – maximum internal iteration allowed

- **dt_min** (*float or None, optional, default None*) – minimum internal time step allowed

class triflow.plugins.schemes.**ROW_general**(*model*,   *alpha*,   *gamma*,   *b*,   *b_pred=None*, *time_stepping=False*, *tol=None*, *max_iter=None*, *dt_min=None*)

Bases: [object](#)

Rosenbrock Wanner class of temporal solvers

The implementation and the different parameters can be found in http://www.digibib.tu-bs.de/?docid=00055262

__call__(*t*, *fields*, *dt*, *pars*, *hook=<function ROW_general.<lambda>>*)

Perform a step of the solver: took a time and a system state as a triflow Fields container and return the next time step with updated container.

Parameters

- **t** (*float*) – actual time step

- **fields** (*triflow.Fields*) – actual system state in a triflow Fields

- **dt** (*float*) – temporal step-size

- **pars** (*dict*) – physical parameters of the model

- **hook** (*callable, optional*) – any callable taking the actual time, fields and parameters and return modified fields and parameters. Will be called every internal time step and can be used to include time dependent or conditionnal parameters, boundary conditions...

- **container** –

Returns tuple – updated time and fields container

Return type *t*, *fields*

Raises

- NotImplementedError

- raised if a time stepping is requested but the scheme do not provide the b predictor coefficients.

- ValueError

- raised if time_stepping is True and tol is not provided.

class triflow.plugins.schemes.**Theta**(*model*, *theta=1*, *solver=<function spsolve>*)

Bases: [object](#)

Simple theta-based scheme where theta is a weight if theta = 0, the scheme is a forward-euler scheme if theta = 1, the scheme is a backward-euler scheme if theta = 0.5, the scheme is called a Crank-Nicolson scheme

Parameters

- **model** (*triflow.Model*) – triflow Model

- **theta** (*int, optional, default 1*) – weight of the theta-scheme

- **solver**   (*callable, optional, default scipy.sparse.linalg. spsolve*) – method able to solve a Ax = b linear equation with A a sparse matrix. Take A and b as argument and return x.

__call__ (*t*, *fields*, *dt*, *pars*, *hook=<function Theta.<lambda>>*)

Perform a step of the solver: took a time and a system state as a triflow Fields container and return the next time step with updated container.

> **Parameters**
>
> - **t** (*float*) – actual time step
> - **fields** (*triflow.Fields*) – actual system state in a triflow Fields container
> - **dt** (*float*) – temporal step-size
> - **pars** (*dict*) – physical parameters of the model
> - **hook** (*callable, optional*) – any callable taking the actual time, fields and parameters and return modified fields and parameters. Will be called every internal time step and can be used to include time dependent or conditionnal parameters, boundary conditions...
>
> **Returns** **tuple** – updated time and fields container
>
> **Return type** *t*, *fields*

class triflow.plugins.schemes.**scipy_ode** (*model*, *integrator='vode'*, ***integrator_kwargs*)

Bases: `object`

Proxy written around the scipy.integrate.ode class. Give access to all the scpy integrators.

> **Parameters**
>
> - **model** (*triflow.Model*) – triflow Model
> - **integrator** (*str, optional, default 'vode'*) – name of the chosen scipy integration scheme.
> - ****integrator_kwargs** – extra arguments provided to the scipy integration scheme.

__call__ (*t*, *fields*, *dt*, *pars*, *hook=<function scipy_ode.<lambda>>*)

Perform a step of the solver: took a time and a system state as a triflow Fields container and return the next time step with updated container.

> **Parameters**
>
> - **t** (*float*) – actual time step
> - **fields** (*triflow.Fields*) – actual system state in a triflow Fields
> - **dt** (*float*) – temporal step-size
> - **pars** (*dict*) – physical parameters of the model
> - **hook** (*callable, optional*) – any callable taking the actual time, fields and parameters and return modified fields and parameters. Will be called every internal time step and can be used to include time dependent or conditionnal parameters, boundary conditions...
> - **container** –
>
> **Returns** **tuple** – updated time and fields container
>
> **Return type** *t*, *fields*
>
> **Raises** `RuntimeError` – Description

triflow.plugins.signals module

This module provides a Signal class usefull for time variable boundary conditions.

Available signals:

> - ConstantSignal: just an offset signal
>
> - ForcedSignal: a sinusoidal wave
>
> - WhiteNoise: a noisy signal
>
> - BrownNoise: a noisy signal with Fourrier modes set to 0 for a fraction of the available modes.

class triflow.plugins.signals.**AdditiveSignal**(*signal_a*, *signal_b*, *op*)

> Bases: *triflow.plugins.signals.Signal*

Additive signal. Proxy the different signals and sum their interpolation functions.

class triflow.plugins.signals.**ConstantSignal**(*offset: float*)

> Bases: *triflow.plugins.signals.Signal*

Offset signal

> **Parameters offset** (*float*) – Value of the offset

Examples

Set an offset to a white noise:

```python
>>> import numpy as np
>>> from triflow.plugins import signals
>>> t = np.linspace(0, 1, 1000)
>>> noisy = signals.GaussianWhiteNoise(frequency_sampling=200)
>>> offset = signals.ConstantSignal(offset=1)
>>> noise_with_offset = noisy + offset
>>> noise_without_offset = noise_with_offset – offset
>>> np.isclose(noisy(t), noise_without_offset(t)).all()
True
```

class triflow.plugins.signals.**GaussianBrownNoise**(*frequency_sampling: float*, *frequency_cut: float*, *mean=0*, *std=0.2*, *n=1000*, *filter_std=25*, *filter_window_size=500*, *seed=None*)

> Bases: *triflow.plugins.signals.GaussianWhiteNoise*

Gaussian Brown noise signal, seeded with numpy.random.randn and with frequency cut at specific value.

> **Parameters**
>
> - **frequency_sampling** (*float*) – Frequency sampling (the highest frequency of the white signal spectrum).
>
> - **frequency_cut** (*float*) – Filter frequency cut.
>
> - **mean** (*int, optional, default 1000*) – sampling number of the signal.
>
> - **n** (*int, optional, default 1000*) – sampling number of the signal.
>
> - **seed** (*int or None, optional*) – pseudo-random number generator seed. Signals with same signal_period, sampling number and seed will be similar.

Examples

White signal with frequencies cut after 50 Hz:

```
>>> from triflow.plugins import signals
>>> brown_signal = signals.GaussianBrownNoise(frequency_sampling=200,
...                                           frequency_cut=50)
>>> spectrum_frequencies, spectrum_density = brown_signal.fourrier_spectrum
>>> np.isclose(spectrum_density[spectrum_frequencies > 50], 0, atol=1E-3).all()
True
```

class triflow.plugins.signals.**GaussianWhiteNoise**(*frequency_sampling*, *mean=0*, *std=0.2*, *n=1000*, *seed=None*)

Bases: *triflow.plugins.signals.Signal*

Gaussian White noise signal, seeded with numpy.random.randn

> **Parameters**
>
> - **frequency_sampling** (*float*) – Frequency sampling (the highest frequency of the white signal spectrum).
> - **mean** (*int, optional, default 1000*) – sampling number of the signal.
> - **n** (*int, optional, default 1000*) – sampling number of the signal.
> - **seed** (*int or None, optional*) – pseudo-random number generator seed. Signals with same signal_period, sampling number and seed will be similar.

Examples

Simple white signal with a 200 Hz frequency sampling:

```
>>> import numpy as np
>>> from triflow.plugins import signals
>>> signal = signals.GaussianWhiteNoise(frequency_sampling=200)
```

Forcing the pseudo-random-number generator seed for reproductible signal:

```
>>> import numpy as np
>>> from triflow.plugins import signals
>>> t = np.linspace(0, 10, 1000)
>>> signal1 = signals.GaussianWhiteNoise(frequency_sampling=200, seed=50)
>>> signal2 = signals.GaussianWhiteNoise(frequency_sampling=200, seed=50)
>>> np.isclose(signal1(t) - signal2(t), 0).all()
True
>>> signal3 = signals.GaussianWhiteNoise(frequency_sampling=200)
>>> np.isclose(signal1(t) - signal3(t), 0).all()
False
```

class triflow.plugins.signals.**Signal**(*signal_period: float*, *n: int = 1000*, ***kwargs*)

Bases: *object*

Base class for signal object.

> **Parameters**
>
> - **signal_period** (*float*) – period of the signal.
> - **n** (*int, optional*) – sampling number of the signal.
> - ****kwargs** – extra arguments provided to the custom signal template.
>
> **Raises**

- `NotImplementedError`

- this class is not supposed to be used by the user. If initialized directly, it will raise a NotImplementedError.

__add__(*other_signal*)

> **Parameters other_signal** (`triflow.plugins.signal.Signal`) – second signal to which this one will be added.
>
> **Returns** added signal.
>
> **Return type** triflow.plugins.signal.AdditiveSignal

__call__(*t: float*)

return the signal value for the time t.

> **Parameters t** (*float*) – time where the signal is evaluated.
>
> **Returns** amplitude of the signal a the time t.
>
> **Return type** float

__sub__(*other_signal*)

> **Parameters other_signal** (`triflow.plugins.signal.Signal`) – second signal to which this one will be added.
>
> **Returns** added signal.
>
> **Return type** triflow.plugins.signal.AdditiveSignal

fourrier_spectrum

param *args, **kwargs: extra arguments provided to the periodogram function.

> **Returns tuple of numpy.ndarray**
>
> **Return type** fourrier modes and power density obtained with the scipy.signal.periodogram function.

class triflow.plugins.signals.**SinusoidalSignal**(*frequency: float, amplitude: float, phase: float = 0, n: int = 1000*)

Bases: `triflow.plugins.signals.Signal`

Simple sinusoidal signal

> **Parameters**
>
> - **frequency** (*float*) – frequency of the signal
>
> - **amplitude** (*float*) – amplitude of the signal
>
> - **phase** (*float, optional*) – phase of the signam
>
> - **n** (*int, optional*) – number of sample for 2 period of the signal

Examples

We generate a 5 Hz signal with an amplitude of 0.5, and we check the signal.

```
>>> from triflow.plugins import signals
>>> t = np.linspace(0, 100, 10000)
>>> signal = signals.SinusoidalSignal(frequency=5, amplitude=.5)
>>> spectrum_frequencies, spectrum_density  = signal.fourrier_spectrum
>>> print(f"Amplitude: {signal(t).max():g}")
```

```
Amplitude: 0.499999
>>> print(f"main mode: {spectrum_frequencies[spectrum_density.argmax()]:g} Hz")
main mode: 4.995 Hz
```

The class give a warning if the number of sample is too low and lead to aliasing. You can try it with

```
>>> import logging
>>> logger = logging.getLogger('triflow.plugins.signals')
>>> logger.handlers = []
>>> logger.addHandler(logging.StreamHandler())
>>> logger.setLevel('INFO')
>>> aliased_signal = signals.SinusoidalSignal(5, 1, n=30)
```

Module contents

5.1.2 Module contents

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

Symbols

A

B

C

D

F

G

I

J

L

M

P

R

S