fluidsim Documentation

Release 0.0.3a0

Pierre Augier

December 17, 2015

Contents

FluidSim is a framework for studying fluid dynamics with numerical simulations using Python. It is part of the wider project FluidDyn.

The package is still in a planning stage so it is still pretty unstable and many of its planned features have not yet been implemented.

FluidSim provides object-oriented libraries to develop quite simple solvers (mainly using pseudo-spectral methods) by writing mainly Python code. The result should be quite efficient compared to a pure Fortran or C++ code since most of the time-consuming tasks are performed by quite optimized compiled functions (to be better quantified).

An advantage is that to run simulations and analyze the results, the users communicate (possibly interactively) with the machine through Python, which is nowadays among the best languages to do these tasks. Moreover, it should be much simpler than with pure Fortran or C++ codes to add any complicate analysis. For example, it should be very simple and quick to write a solver for adjoin equations.

At this stage, just few solvers have been written, but at least FluidSim can solve these equations:

- Incompressible Navier-Stokes equations in a two-dimensional periodic space,
- One-layer shallow-water equations in a two-dimensional periodic space,
- ...

CHAPTER 1

User Guide

1.1 Installation

FluidSim is part of the FluidDyn project. Some issues regarding the installation of Python packages are discussed in the main documentation of the project.

1.1.1 Dependencies

- FFTW3 (and some modules take advantage of libfftw3_mpi, but this one is optional) and on the python package pyfftw,

The first thing to do before installing FluidSim is to installed these libraries (in contrast, the Python packages should automatically be installed by the installer)!

- optionally, mpi4py (which depends on a MPI implementation).

1.1.2 Install commands

FluidSim can be installed by running the following commands:

```
hg clone https://bitbucket.org/fluiddyn/fluidsim
cd fluidsim
python setup.py develop
```

Installation with Pip should also work:

```
pip install fluidsim
```

1.1.3 Run the tests

You can run some unit tests by running `make tests` from the root directory or `python -m unittest discover` from the root directory or from any of the "test" directories.

1.2 Tutorials

Most of these tutorials have been produced by Ipython notebook.

```
from __future__ import print_function
%matplotlib inline
import fluidsim
```

1.2.1 Tutorial: running a simulation (user perspective)

In this tutorial, I'm going to show how to run a simple simulation with a solver that solves the 2 dimensional Navier-Stokes equations. I'm also going to present some useful concepts and objects used in FluidSim.

A minimal simulation

Fisrt, let's see what is needed to run a very simple simulation. For the initialization (with default parameters):

```
from fluidsim.solvers.ns2d.solver import Simul
params = Simul.create_default_params()
sim = Simul(params)
```

```
**********************************
Program FluidDyn

solver NS2D, RK4 and sequential,
type fft: FFTWCY
nx =     48 ; ny =     48
Lx = 8. ; Ly = 8.
path_run =
/home/pierre/Sim_data/NS2D_L=8.x8._48x48_2015-06-26_13-41-13
init_fields.type: constant
Initialization outputs:
<class `fluidsim.base.output.increments.Increments'> increments
<class `fluidsim.base.output.phys_fields.PhysFieldsBase'> phys_fields
<class `fluidsim.solvers.ns2d.output.spectra.SpectraNS2D'> spectra
<class `fluidsim.solvers.ns2d.output.spatial_means.SpatialMeansNS2D'> spatial_means
<class `fluidsim.solvers.ns2d.output.spect_energy_budget.SpectralEnergyBudgetNS2D'> spect_e

Memory usage at the end of init. (equiv. seq.): 71.0546875 Mo
Size of state_fft (equiv. seq.): 0.0192 Mo
```

And then to run the simulation:

```
sim.time_stepping.start()
```

```
**********************************
Beginning of the computation
save state_phys in file state_phys_t=000.000_it=0.hd5
    compute until t =         10
it =      0 ; t =          0 ; deltat  =   0.083333
              energy = 0.000e+00 ; Delta energy = +0.000e+00

it =      6 ; t =    1.08333 ; deltat  =        0.2
              energy = 0.000e+00 ; Delta energy = +0.000e+00
              estimated remaining duration =     0.194 s
it =     12 ; t =    2.28333 ; deltat  =        0.2
              energy = 0.000e+00 ; Delta energy = +0.000e+00
              estimated remaining duration =     0.128 s
```

```
it =     17 ; t =    3.28333 ; deltat  =        0.2
              energy = 0.000e+00 ; Delta energy = +0.000e+00
              estimated remaining duration =     0.149 s
it =     22 ; t =    4.28333 ; deltat  =        0.2
              energy = 0.000e+00 ; Delta energy = +0.000e+00
              estimated remaining duration =    0.0996 s
it =     27 ; t =    5.28333 ; deltat  =        0.2
              energy = 0.000e+00 ; Delta energy = +0.000e+00
              estimated remaining duration =     0.113 s
it =     32 ; t =    6.28333 ; deltat  =        0.2
              energy = 0.000e+00 ; Delta energy = +0.000e+00
              estimated remaining duration =    0.0741 s
it =     37 ; t =    7.28333 ; deltat  =        0.2
              energy = 0.000e+00 ; Delta energy = +0.000e+00
              estimated remaining duration =    0.0435 s
it =     43 ; t =    8.48333 ; deltat  =        0.2
              energy = 0.000e+00 ; Delta energy = +0.000e+00
              estimated remaining duration =    0.0202 s
it =     49 ; t =    9.68333 ; deltat  =        0.2
              energy = 0.000e+00 ; Delta energy = +0.000e+00
              estimated remaining duration =   0.00476 s
Computation completed in 0.194266 s
path_run =
/home/pierre/Sim_data/NS2D_L=8.x8._48x48_2015-06-26_13-41-13
save state_phys in file state_phys_t=010.083_it=51.hd5
```

In the following, we are going to understand these 4 lines of code... But first let's clean-up by deleting the result directory of this tiny example simulation:

```
import shutil
shutil.rmtree(sim.output.path_run)
```

Importing a solver

The first line imports a "Simulation" class from a "solver" module. Any solver module has to provide a class called "Simul". We have already seen that the Simul class can be imported like this:

```
from fluidsim.solvers.ns2d.solver import Simul
```

but there is another convenient way to import it from a string:

```
Simul = fluidsim.import_simul_class_from_key('ns2d')
```

Create an instance of the class Parameters

The next step is to create an object `params` from the information contained in the class `Simul`:

```
params = Simul.create_default_params()
```

The object `params` is an instance of the class *`fluidsim.base.params.Parameters`* (which inherits from fluiddyn.util.paramcontainer.ParamContainer). It is usually a quite complex object containing many attributes. In this case, it contains many parameters. To print them, the normal way would be to use the tab-completion of Ipython, i.e. to type "*params.*" and press on the tab key. Here, I can not do that so I'm going to use a command that produce a list with the interesting attributes. If you don't understand this command, you should have a look at the section on list comprehensions of the official Python tutorial):

```
[attr for attr in dir(params) if not attr.startswith('_')]
```

```
['FORCING',
 'NEW_DIR_RESULTS',
 'ONLY_COARSE_OPER',
 'beta',
 'forcing',
 'init_fields',
 'nu_2',
 'nu_4',
 'nu_8',
 'nu_m4',
 'oper',
 'output',
 'short_name_type_run',
 'time_stepping']
```

and some useful functions (whose names all start with _ in order to be hidden in Ipython):

```
[attr for attr in dir(params) if attr.startswith('_') and not attr.startswith('__')]
```

```
['_attribs',
 '_load_from_elemxml',
 '_load_from_hdf5_file',
 '_load_from_hdf5_objet',
 '_load_from_xml_file',
 '_make_dict',
 '_make_element_xml',
 '_make_xml_text',
 '_print_as_xml',
 '_save_as_hdf5',
 '_save_as_xml',
 '_set_as_child',
 '_set_attrib',
 '_set_attribs',
 '_set_child',
 '_set_internal_attr',
 '_tag',
 '_tag_children']
```

Some of the attributes of `params` are simple Python objects and others can be other *`fluidsim.base.params.Parameters`*:

```
print(type(params.nu_2))
print(type(params.output))
```

```
<type 'float'>
<class 'fluidsim.base.params.Parameters'>
```

```
[attr for attr in dir(params.output) if not attr.startswith('_')]
```

```
['HAS_TO_SAVE',
 'ONLINE_PLOT_OK',
 'increments',
 'period_refresh_plots',
 'periods_plot',
 'periods_print',
 'periods_save',
```

```
 'phys_fields',
 'spatial_means',
 'spect_energy_budg',
 'spectra',
 'sub_directory']
```

We see that the object `params` contains a tree of parameters. This tree can be represented as xml code:

```
print(params)
```

```
<fluidsim.base.params.Parameters object at 0x7f8c52e034d0>

<params ONLY_COARSE_OPER="False" short_name_type_run="" beta="0.0" nu_2="0.0"
        NEW_DIR_RESULTS="True" nu_4="0.0" nu_8="0.0" FORCING="False"
        nu_m4="0.0">
  <oper type_fft="FFTWCY" nx="48" ny="48" coef_dealiasing="0.6666666666666666"
        TRANSPOSED_OK="True" Lx="8" Ly="8"/>

  <init_fields available_types="['from_file', 'noise', 'constant', 'jet',
               'manual', 'dipole', 'from_simul']" type="constant">
    <from_file path=""/>

    <noise length="0" velo_max="1.0"/>

    <constant value="1.0"/>

  </init_fields>

  <forcing nkmax_forcing="5" nkmin_forcing="4" key_forced="rot_fft"
           available_types="['proportional', 'random']" type=""
           forcing_rate="1">
    <random type_normalize="2nd_degree_eq"
            time_correlation="based_on_forcing_rate"/>

  </forcing>

  <time_stepping type_time_scheme="RK4" it_end="10" USE_CFL="True" deltat0="0.2"
                 t_end="10.0" USE_T_END="True"/>

  <output period_refresh_plots="1" HAS_TO_SAVE="True" ONLINE_PLOT_OK="True"
          sub_directory="">
    <periods_plot phys_fields="0"/>

    <periods_print print_stdout="1.0"/>

    <increments HAS_TO_PLOT_SAVED="False"/>

    <spectra HAS_TO_PLOT_SAVED="False"/>

    <spatial_means HAS_TO_PLOT_SAVED="False"/>

    <spect_energy_budg HAS_TO_PLOT_SAVED="False"/>

    <phys_fields field_to_plot="rot" file_with_it="False"/>

    <periods_save spect_energy_budg="0" spatial_means="0" spectra="0"
                  increments="0" phys_fields="0"/>
```

```
  </output>

</params>
```

Set the parameters for your simulation

The user can change any parameters

```
params.nu_2 = 1e-3
params.FORCING = False

params.init_fields.type = 'noise'

params.output.periods_save.spatial_means = 1.
params.output.periods_save.spectra = 1.
```

but it is impossible to create accidentally a parameter which is actually not used:

```
try:
    params.this_param_does_not_exit = 10
except AttributeError as e:
    print('AttributeError:', e)
```

```
AttributeError: this_param_does_not_exit is not already set in params.
The attributes are: set(['ONLY_COARSE_OPER', 'short_name_type_run', 'beta', 'nu_2', 'NEW_DIR_RESULTS
To set a new attribute, use _set_attrib or _set_attribs.
```

This behaviour is much safer than using a text file or a python file for the parameters. In order to discover the different parameters for a solver, create the `params` object containing the default parameters in Ipython (`params = Simul.create_default_params()`), print it and use the auto-completion (for example writting `params.` and pressing on the tab key).

Instantiate a simulation object

The next step is to create a simulation object (an instance of the class solver.Simul) with the parameters in `params`:

```
sim = Simul(params)
```

```
**********************************
Program FluidDyn

solver NS2D, RK4 and sequential,
type fft: FFTWCY
nx =     48 ; ny =     48
Lx = 8. ; Ly = 8.
path_run =
/home/pierre/Sim_data/NS2D_L=8.x8._48x48_2015-06-26_13-41-14
init_fields.type: noise
Initialization outputs:
<class `fluidsim.base.output.increments.Increments'> increments
<class `fluidsim.base.output.phys_fields.PhysFieldsBase'> phys_fields
<class `fluidsim.solvers.ns2d.output.spectra.SpectraNS2D'> spectra
<class `fluidsim.solvers.ns2d.output.spatial_means.SpatialMeansNS2D'> spatial_means
<class `fluidsim.solvers.ns2d.output.spect_energy_budget.SpectralEnergyBudgetNS2D'> spect_e
```

```
Memory usage at the end of init. (equiv. seq.): 73.3515625 Mo
Size of state_fft (equiv. seq.): 0.0192 Mo
```

which initializes everything needed to run the simulation. The object `sim` has a limited number of attributes:

```
[attr for attr in dir(sim) if not attr.startswith('_')]
```

```
['InfoSolver',
 'compute_freq_diss',
 'create_default_params',
 'info',
 'info_solver',
 'init_fields',
 'name_run',
 'oper',
 'output',
 'params',
 'state',
 'tendencies_nonlin',
 'time_stepping']
```

In the tutorial Understand how works FluidSim, we will see what are all these attributes.

The object `sim.info` is a `fluiddyn.util.paramcontainer.ParamContainer` which contains all the information on the solver (in `sim.info.solver`) and on specific parameters for this simulation (in `sim.info.solver`):

```
print(sim.info.__class__)
print([attr for attr in dir(sim.info) if not attr.startswith('_')])
```

```
<class 'fluiddyn.util.paramcontainer.ParamContainer'>
['params', 'solver']
```

```
sim.info.solver is sim.info_solver
```

```
True
```

```
sim.info.params is sim.params
```

```
True
```

```
print(sim.info.solver)
```

```
<fluidsim.solvers.ns2d.solver.InfoSolverNS2D object at 0x7f8c52e03090>

<solver class_name="Simul" module_name="fluidsim.solvers.ns2d.solver"
        short_name="NS2D">
  <classes>
    <Operators class_name="OperatorsPseudoSpectral2D"
               module_name="fluidsim.operators.operators"/>

    <InitFields class_name="InitFieldsNS2D"
                module_name="fluidsim.solvers.ns2d.init_fields">
      <classes>
        <from_file class_name="InitFieldsFromFile"
                   module_name="fluidsim.base.init_fields"/>

        <noise class_name="InitFieldsNoise"
               module_name="fluidsim.solvers.ns2d.init_fields"/>
```

```
      <constant class_name="InitFieldsConstant"
                module_name="fluidsim.base.init_fields"/>

      <jet class_name="InitFieldsJet"
           module_name="fluidsim.solvers.ns2d.init_fields"/>

      <manual class_name="InitFieldsManual"
              module_name="fluidsim.base.init_fields"/>

      <dipole class_name="InitFieldsDipole"
              module_name="fluidsim.solvers.ns2d.init_fields"/>

      <from_simul class_name="InitFieldsFromSimul"
                  module_name="fluidsim.base.init_fields"/>

    </classes>

  </InitFields>

  <TimeStepping class_name="TimeSteppingPseudoSpectral"
                module_name="fluidsim.base.time_stepping.pseudo_spect_cy"/>

  <State keys_linear_eigenmodes="['rot_fft']" keys_state_fft="['rot_fft']"
         class_name="StateNS2D" keys_phys_needed="['rot']"
         keys_state_phys="['ux', 'uy', 'rot']"
         module_name="fluidsim.solvers.ns2d.state" keys_computable="[]"/>

  <Output class_name="Output" module_name="fluidsim.solvers.ns2d.output">
    <classes>
      <PrintStdOut class_name="PrintStdOutNS2D"
                   module_name="fluidsim.solvers.ns2d.output.print_stdout"/>

      <increments class_name="Increments"
                  module_name="fluidsim.base.output.increments"/>

      <PhysFields class_name="PhysFieldsBase"
                  module_name="fluidsim.base.output.phys_fields"/>

      <Spectra class_name="SpectraNS2D"
               module_name="fluidsim.solvers.ns2d.output.spectra"/>

      <spatial_means class_name="SpatialMeansNS2D"
                     module_name="fluidsim.solvers.ns2d.output.spatial_means"/>

      <spect_energy_budg class_name="SpectralEnergyBudgetNS2D"
                         module_name="fluidsim.solvers.ns2d.output.spect_energy_budget"/>

    </classes>

  </Output>

  <Forcing class_name="ForcingNS2D"
           module_name="fluidsim.solvers.ns2d.forcing">
    <classes>
      <proportional class_name="Proportional"
                    module_name="fluidsim.base.forcing.specific"/>

      <random class_name="TimeCorrelatedRandomPseudoSpectral"
```

```
                module_name="fluidsim.base.forcing.specific"/>

      </classes>

    </Forcing>

  </classes>

</solver>
```

We see that a solver is defined by the classes it uses for some tasks. The tutorial Understand how works FluidSim is meant to explain how.

Run the simulation

We can now start the time stepping. Since `params.time_stepping.USE_T_END is True`, it should loop until `sim.time_stepping.t` is equal or larger than `params.time_stepping.t_end = 10`.

```
sim.time_stepping.start()
```

```
**********************************
Beginning of the computation
save state_phys in file state_phys_t=000.000_it=0.hd5
    compute until t =         10
it =      0 ; t =          0 ; deltat  =   0.097144
               energy = 9.159e-02 ; Delta energy = +0.000e+00

it =     11 ; t =    1.09076 ; deltat  =    0.10203
               energy = 9.061e-02 ; Delta energy = -9.864e-04
               estimated remaining duration =     0.216 s
it =     21 ; t =     2.1292 ; deltat  =     0.1043
               energy = 8.968e-02 ; Delta energy = -9.244e-04
               estimated remaining duration =     0.265 s
it =     31 ; t =    3.16728 ; deltat  =    0.10186
               energy = 8.878e-02 ; Delta energy = -9.062e-04
               estimated remaining duration =     0.212 s
it =     41 ; t =    4.17421 ; deltat  =   0.099527
               energy = 8.792e-02 ; Delta energy = -8.558e-04
               estimated remaining duration =     0.201 s
it =     52 ; t =    5.25129 ; deltat  =   0.099822
               energy = 8.704e-02 ; Delta energy = -8.819e-04
               estimated remaining duration =     0.151 s
it =     62 ; t =    6.29295 ; deltat  =    0.10683
               energy = 8.622e-02 ; Delta energy = -8.137e-04
               estimated remaining duration =     0.138 s
it =     72 ; t =    7.35239 ; deltat  =    0.10687
               energy = 8.544e-02 ; Delta energy = -7.870e-04
               estimated remaining duration =    0.0827 s
it =     82 ; t =    8.44874 ; deltat  =    0.10722
               energy = 8.466e-02 ; Delta energy = -7.756e-04
               estimated remaining duration =    0.0502 s
it =     92 ; t =    9.49721 ; deltat  =    0.10258
               energy = 8.395e-02 ; Delta energy = -7.088e-04
               estimated remaining duration =    0.0158 s
```

```
Computation completed in 0.344521 s
path_run =
/home/pierre/Sim_data/NS2D_L=8.x8._48x48_2015-06-26_13-41-14
save state_phys in file state_phys_t=010.010_it=97.hd5
```

Analyze the output

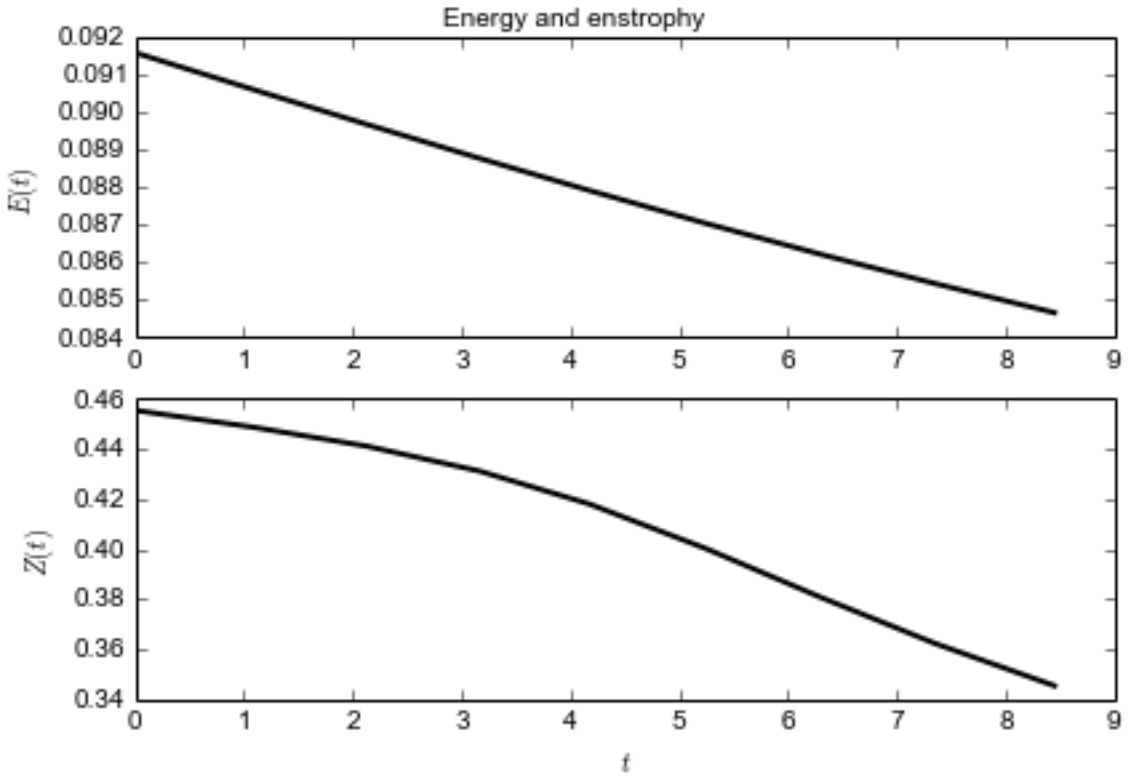
Let's see what we can do with the object `sim.output`. What are its attributes?

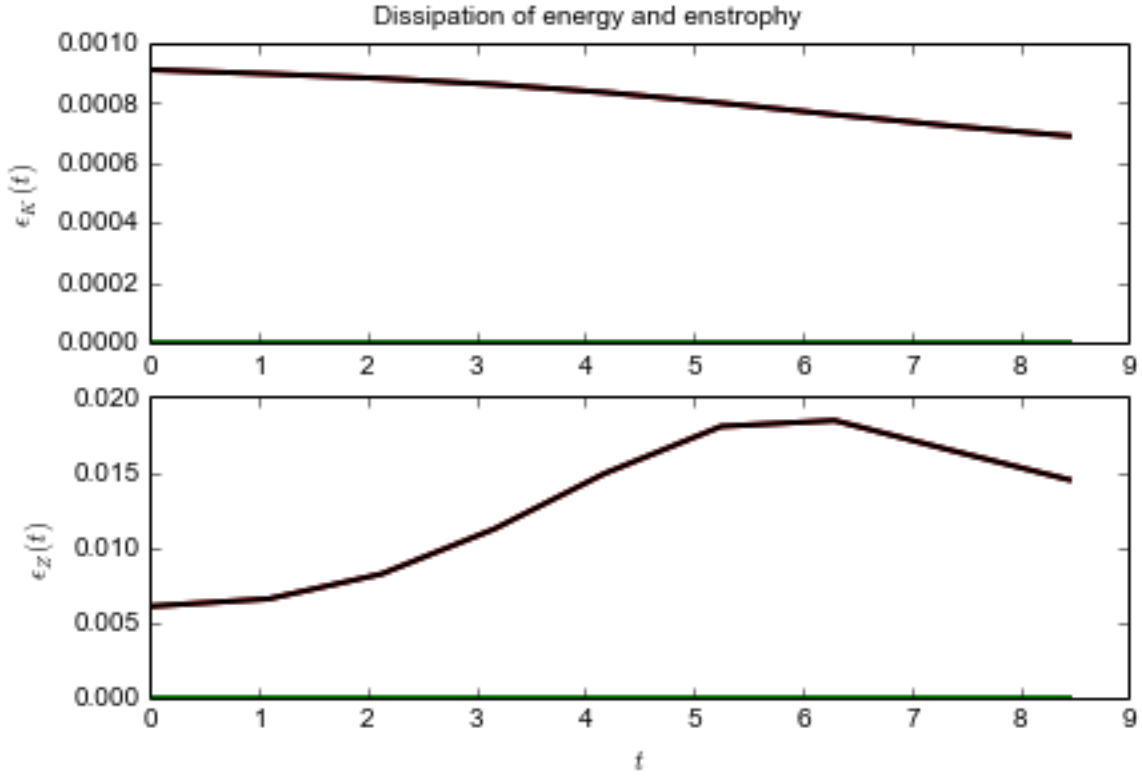
```
[attr for attr in dir(sim.output) if not attr.startswith('_')]
```

```
['compute_energy',
 'compute_energy_fft',
 'compute_enstrophy',
 'compute_enstrophy_fft',
 'create_list_for_name_run',
 'end_of_simul',
 'figure_axe',
 'has_been_initialized_with_state',
 'has_to_save',
 'increments',
 'init_with_initialized_state',
 'init_with_oper_and_state',
 'name_run',
 'name_solver',
 'one_time_step',
 'oper',
 'params',
 'path_run',
 'phys_fields',
 'print_size_in_Mo',
 'print_stdout',
 'sim',
 'spatial_means',
 'spect_energy_budg',
 'spectra',
 'sum_wavenumbers']
```

Many of these objects (`print_stdout`, `phys_fields`, `spatial_means`, `spect_energy_budg`, `spectra`, ...) were used during the simulation to save outputs. They can also load the data and produce some simple plots. For example, to display the time evolution of spatially averaged quantities (here the energy, the entrophy and their dissipation rate):

```
sim.output.spatial_means.plot()
```





Finally we remove the directory of this example simulation...

```
shutil.rmtree(sim.output.path_run)
```

```
from __future__ import print_function
%matplotlib inline
```

1.2.2 Tutorial: understand how works FluidSim

A goal of FluidSim is to be as simple as possible to allow anyone knowing a little bit of Python to understand how it works internally. For this tutorial, it is assumed that the reader knows how to run simulations with FluidSim. If it is not the case, first read the tutorial running a simulation (user perspective).

A class to organized parameters

First, we need to present the important class fluiddyn.util.paramcontainer.ParamContainer used to contain information.

```
from fluiddyn.util.paramcontainer import ParamContainer
params = ParamContainer(tag='params')
params._set_attribs({'a0': 1, 'a1': 1})
params._set_attrib('a2', 1)
params._set_child('child0', {'a0': 1})
params.a2 = 2
params.child0.a0 = 'other option'
```

A `ParamContainer` can be represented as xml

```
params
```

```
<fluiddyn.util.paramcontainer.ParamContainer object at 0x7fc36a7a61d0>

<params a1="1" a0="1" a2="2">
  <child0 a0="other option"/>

</params>
```

FluidSim uses instances of this class to store the information of a particular solver and the parameters of a particular simulation.

The Simul classes and the default parameters

The first step to run a simulation is to import a Simul class from a solver module, for example:

```
from fluidsim.solvers.ns2d.solver import Simul
```

Any solver module has to define a class called Simul which has to have some important attributes:

```
[name for name in dir(Simul) if not name.startswith('__')]
```

```
['InfoSolver',
 '_complete_params_with_default',
 'compute_freq_diss',
 'create_default_params',
 'info_solver',
 'tendencies_nonlin']
```

The first attribute `InfoSolver` is a class deriving from `ParamContainer`. This class is usually defined in the *solver* module. It is used during the instantiation of the Simul object to produce a `ParamContainer` containing a description of the solver, in practice the names and the modules of the classes used for the different tasks that need to be performed during the simulation.

There are also four other functions. `compute_freq_diss()` and `tendencies_nonlin()` are used during the simulation and describe the equations that are solved.

`create_default_params()` and `_complete_params_with_default()` are used to produce the *ParamContainer* containing the default parameters for a simulation:

```
params = Simul.create_default_params()
```

During the creation of *params*, the class `InfoSolver` has been used to create a `ParamContainer` named *info_solver*:

```
Simul.info_solver
```

```
<fluidsim.solvers.ns2d.solver.InfoSolverNS2D object at 0x7fc36a79a7d0>

<solver class_name="Simul" module_name="fluidsim.solvers.ns2d.solver"
        short_name="NS2D">
  <classes>
    <Operators class_name="OperatorsPseudoSpectral2D"
               module_name="fluidsim.operators.operators"/>

    <InitFields class_name="InitFieldsNS2D"
                module_name="fluidsim.solvers.ns2d.init_fields">
      <classes>
        <from_file class_name="InitFieldsFromFile"
                   module_name="fluidsim.base.init_fields"/>

        <noise class_name="InitFieldsNoise"
               module_name="fluidsim.solvers.ns2d.init_fields"/>

        <constant class_name="InitFieldsConstant"
                  module_name="fluidsim.base.init_fields"/>

        <jet class_name="InitFieldsJet"
             module_name="fluidsim.solvers.ns2d.init_fields"/>

        <manual class_name="InitFieldsManual"
                module_name="fluidsim.base.init_fields"/>

        <dipole class_name="InitFieldsDipole"
                module_name="fluidsim.solvers.ns2d.init_fields"/>

        <from_simul class_name="InitFieldsFromSimul"
                    module_name="fluidsim.base.init_fields"/>

      </classes>

    </InitFields>

    <TimeStepping class_name="TimeSteppingPseudoSpectral"
                  module_name="fluidsim.base.time_stepping.pseudo_spect_cy"/>

    <State keys_linear_eigenmodes="['rot_fft']" keys_state_fft="['rot_fft']"
           class_name="StateNS2D" keys_phys_needed="['rot']"
```

```
          keys_state_phys="['ux', 'uy', 'rot']"
          module_name="fluidsim.solvers.ns2d.state" keys_computable="[]"/>

    <Output class_name="Output" module_name="fluidsim.solvers.ns2d.output">
      <classes>
        <PrintStdOut class_name="PrintStdOutNS2D"
                     module_name="fluidsim.solvers.ns2d.output.print_stdout"/>

        <increments class_name="Increments"
                    module_name="fluidsim.base.output.increments"/>

        <PhysFields class_name="PhysFieldsBase"
                    module_name="fluidsim.base.output.phys_fields"/>

        <Spectra class_name="SpectraNS2D"
                 module_name="fluidsim.solvers.ns2d.output.spectra"/>

        <spatial_means class_name="SpatialMeansNS2D"
                       module_name="fluidsim.solvers.ns2d.output.spatial_means"/>

        <spect_energy_budg class_name="SpectralEnergyBudgetNS2D"
                           module_name="fluidsim.solvers.ns2d.output.spect_energy_budget"/>

      </classes>

    </Output>

    <Forcing class_name="ForcingNS2D"
             module_name="fluidsim.solvers.ns2d.forcing">
      <classes>
        <proportional class_name="Proportional"
                      module_name="fluidsim.base.forcing.specific"/>

        <random class_name="TimeCorrelatedRandomPseudoSpectral"
                module_name="fluidsim.base.forcing.specific"/>

      </classes>

    </Forcing>

  </classes>

</solver>
```

We see that this solver uses many classes and that they are organized in tasks ("Operator", "InitFields", "TimeStepping", "State", "Output", "Forcing"). Some first-level classes (for example "Output") have second-level classes ("PrintStdOut", "Spectra", "PhysFields", etc.). Such description of a solver is very general. It is also very conveniant to create a new solver from a similar existing solver.

Every classes can have a class function or a static function `_complete_params_with_default()` that is called when the object containing the default parameters is created.

The objects *params* and *Simul.info_solver* are then used to instantiate the simulation (here with the default parameters for the solver):

```
sim = Simul(params)
```

```
**********************************
Program FluidDyn
```

```
solver NS2D, RK4 and sequential,
type fft: FFTWCY
nx =     48 ; ny =     48
Lx = 8. ; Ly = 8.
path_run =
/home/users/augier3pi/Sim_data/NS2D_L=8.x8._48x48_2015-06-25_18-16-02
init_fields.type: constant
Initialization outputs:
<class `fluidsim.base.output.increments.Increments'> increments
<class `fluidsim.base.output.phys_fields.PhysFieldsBase'> phys_fields
<class `fluidsim.solvers.ns2d.output.spectra.SpectraNS2D'> spectra
<class `fluidsim.solvers.ns2d.output.spatial_means.SpatialMeansNS2D'> spatial_means
<class `fluidsim.solvers.ns2d.output.spect_energy_budget.SpectralEnergyBudgetNS2D'> spect_e

Memory usage at the end of init. (equiv. seq.): 81.3125 Mo
Size of state_fft (equiv. seq.): 0.0192 Mo
```

Let's print the attributes of `sim` that are not class attributes:

```
[name for name in dir(sim) if not name.startswith('_') and name not in dir(Simul)]
```

```
['info',
 'init_fields',
 'name_run',
 'oper',
 'output',
 'params',
 'state',
 'time_stepping']
```

Except `name_run` and `info`, the attributes are instances of the first-level classes defined in `Simul.info_solver`. These different objects have to interact together. We are going to present these different hierarchies of classes but first we come back to the two functions describing the equations in a pseudo-spectral solver.

Description of the solved equations

The functions `Simul.compute_freq_diss()` and `Simul.tendencies_nonlin()` define the solved equations. Looking at the documentation of the solver module *`fluidsim.solvers.ns2d.solver`*, we see that `Simul.tendencies_nonlin()` is defined in this module and that `Simul.compute_freq_diss()` is inherited from the base class *`fluidsim.base.solvers.pseudo_spect.SimulBasePseudoSpectral`*. By clicking on these links, you can look at the documentation and the sources of these functions. The documentation explains how this function define the solved equations. I think the sources are quite clear and can be understood by anyone knowing a little bit of Python for science. Most of the objects involved in these functions are functions or numpy.ndarray.

State classes (`sim.state`)

sim.state is an instance of *`fluidsim.solvers.ns2d.state.StateNS2D`*. It contains numpy.ndarray, actually slightly modified numpy.ndarray named *`fluidsim.base.setofvariables.SetOfVariables`*. This class is used to stack variables together in a single numpy.ndarray.

The state classes are also able to compute other variables from the state of the simulation. It is an interface hidding the actual way the data are stored.

Operator classes (`sim.oper`)

sim.oper is an instance of *fluidsim.operators.operators.OperatorsPseudoSpectral2D*.

It contains the information on the grids (in physical and spectral space) and provides many optimized functions on arrays representing fields on these grids.

It has to be fast! For the two dimensional Fourier pseudo-spectral solvers, it is written in Cython.

TimeStepping classes (`sim.time_stepping`)

sim.time_stepping is an instance of *fluidsim.base.time_stepping.pseudo_spect_cy.TimeSteppingPseudoSpect* which is based on *fluidsim.base.time_stepping.pseudo_spect.TimeSteppingPseudoSpectral* and *fluidsim.base.time_stepping.base.TimeSteppingBase*.

This class contains the functions for the time advancement, i.e. Runge-Kutta functions and the actual loop than increments the time stepping index *sim.time_stepping.it*. The Runge-Kutta functions call the function `sim.tendencies_nonlin()` and modify the state in Fourier space *sim.state.state_fft*.

The loop function also call the function `sim.output.one_time_step()`.

Output classes (`sim.output`)

sim.output is an instance of *fluidsim.solvers.ns2d.output.Output*.

Saving and plotting of anything interesting...

Forcing classes (`sim.forcing`)

sim.forcing is an instance of *fluidsim.solvers.ns2d.forcing.ForcingNS2D*.

If *params.FORCING is True*, it is used in `sim.tendencies_nonlin()` to add the forcing term.

1.3 Examples

1.3.1 Running a simple simulation

This script is an example of how one can run a small simulation with instantaneous plotting. This can be useful for simple tests. It can be launched by the commands `python simul_ns2d.py` (sequencial) and `mpirun -np 8 python simul_ns2d.py` (parallel on 8 processes).

```
from math import pi

from fluidsim.solvers.ns2d.solver import Simul

import fluiddyn as fld

params = Simul.create_default_params()

params.short_name_type_run = 'test'

params.oper.nx = params.oper.ny = nh = 32
params.oper.Lx = params.oper.Ly = Lh = 2 * pi
```

```
delta_x = Lh / nh
params.nu_8 = 2.*params.forcing.forcing_rate**(1./3)*delta_x**8

params.time_stepping.t_end = 2.

params.init_fields.type = 'dipole'

params.FORCING = True
params.forcing.type = 'random'

params.output.sub_directory = 'examples'

params.output.periods_print.print_stdout = 0.25

params.output.periods_save.phys_fields = 1.
params.output.periods_save.spectra = 0.5
params.output.periods_save.spatial_means = 0.05
params.output.periods_save.spect_energy_budg = 0.5
params.output.periods_save.increments = 0.5

params.output.periods_plot.phys_fields = 0.0

params.output.ONLINE_PLOT_OK = True

params.output.spectra.HAS_TO_PLOT_SAVED = True
params.output.spatial_means.HAS_TO_PLOT_SAVED = True
params.output.spect_energy_budg.HAS_TO_PLOT_SAVED = True
params.output.increments.HAS_TO_PLOT_SAVED = True

params.output.phys_fields.field_to_plot = 'rot'

sim = Simul(params)

sim.output.phys_fields.plot()
sim.time_stepping.start()
sim.output.phys_fields.plot()

fld.show()
```

Of course the plots slow down the simulation, so for larger simulation we just remove the plot command from the script, which gives:

```
from math import pi

from fluidsim.solvers.ns2d.solver import Simul

params = Simul.create_default_params()

params.short_name_type_run = 'test'

params.oper.nx = params.oper.ny = nh = 32
params.oper.Lx = params.oper.Ly = Lh = 2 * pi

delta_x = Lh / nh
params.nu_8 = 2.*params.forcing.forcing_rate**(1./3)*delta_x**8

params.time_stepping.t_end = 2.
```

```
params.init_fields.type = 'dipole'

params.FORCING = True
params.forcing.type = 'random'

params.output.sub_directory = 'examples'

params.output.periods_print.print_stdout = 0.25

params.output.periods_save.phys_fields = 1.
params.output.periods_save.spectra = 0.5
params.output.periods_save.spatial_means = 0.05
params.output.periods_save.spect_energy_budg = 0.5
params.output.periods_save.increments = 0.5

params.output.periods_plot.phys_fields = 0.0

sim = Simul(params)

sim.time_stepping.start()
```

To submit the simulation on a cluster (here on one node), just run this tiny script:

```
from fluiddyn.clusters.legi import Calcul3 as Cluster
cluster = Cluster()


cluster.submit_script(
    'simul_ns2d.py', name_run='fld_example',
    nb_cores_per_node=cluster.nb_cores_per_node)
```

CHAPTER 2

Modules Reference

fluidsim.base	Base classes for writing solvers
fluidsim.operators	Numerical operators
fluidsim.solvers	Particular solvers
fluidsim.util	Utilities

2.1 fluidsim.base

2.1.1 Base classes for writing solvers

Provides:

solvers	Base simulations (*fluidsim.base.solvers*)
params	Information on a solver (*fluidsim.base.params*)
setofvariables	Variable container (*fluidsim.base.setofvariables*)
state	State of the variables (*fluidsim.base.state*)
init_fields	Initialisation of the fields (*fluidsim.base.init_fields*)
time_stepping	Time stepping (*fluidsim.base.time_stepping*)
output	Output (*fluidsim.base.output*)
forcing	Forcing schemes (*fluidsim.base.forcing*)
preprocess	Preprocessing of parameters (*fluidsim.base.preprocess*)

fluidsim.base.solvers

Base simulations (`fluidsim.base.solvers`)

Provides:

base	Base solver (*fluidsim.base.solvers.base*)
pseudo_spect	Base solver (*fluidsim.base.solvers.pseudo_spect*)
finite_diff	

fluidsim.base.solvers.base

Base solver (`fluidsim.base.solvers.base`) Provides:

class fluidsim.base.solvers.base.**InfoSolverBase**(***kargs*)
Bases: fluiddyn.util.paramcontainer.ParamContainer

Contain the information on a solver.

import_classes()
Import the classes and return a dictionary.

class fluidsim.base.solvers.base.**SimulBase**(*params*)
Bases: object

Represent a solver.

This is the main base class which is inherited by the other simulation classes.

A *SimulBase* object contains at least one object of the classes:

- *fluidsim.base.params.Parameters*
- fluidsim.base.time_stepping.TimeSteppingBase
- *fluidsim.operators.operators.Operators*
- *fluidsim.base.state.StateBase*

Parameters **params** : *fluidsim.base.params.Parameters*

Parameters for the simulation.

info_solver : fluidsim.base.solvers.info_base.InfoSolverBase

Information about the particular solver.

InfoSolver
alias of *InfoSolverBase*

static _complete_params_with_default(*params*)
A static method used to complete the *params* container.

tendencies_nonlin(*variables=None*)
Return a null SetOfVariables object.

Functions

Classes

Simul	alias of *SimulBase*
SimulBase(params)	Represent a solver.

fluidsim.base.solvers.pseudo_spect

Base solver (fluidsim.base.solvers.pseudo_spect) This module provides two base classes that can be used to define pseudo-spectral solvers.

class fluidsim.base.solvers.pseudo_spect.**InfoSolverPseudoSpectral**(***kargs*)
Bases: fluidsim.base.solvers.info_base.InfoSolverBase

Contain the information on a base pseudo-spectral 2D solver.

_init_root()
Init. *self* by writting the information on the solver.

The first-level classes for this base solver are:

- *fluidsim.base.solvers.pseudo_spect.SimulBasePseudoSpectral*
- *fluidsim.base.state.StatePseudoSpectral*
- *fluidsim.base.time_stepping.pseudo_spect_cy.TimeSteppingPseudoSpectral*
- *fluidsim.operators.operators.OperatorsPseudoSpectral2D*

class fluidsim.base.solvers.pseudo_spect.**SimulBasePseudoSpectral**(*params*)
Bases: *fluidsim.base.solvers.base.SimulBase*

Pseudo-spectral base solver.

InfoSolver
alias of *InfoSolverPseudoSpectral*

static _complete_params_with_default(*params*)
Complete the *params* container (static method).

compute_freq_diss()
Compute the dissipation frequency.

Use the *self.params.nu_...* parameters to compute an array containing the dissipation frequency as a function of the wavenumber.

The governing equations for a pseudo-spectral solver can be written as

$$\partial_t S = N(S) - \sigma(k) S,$$

where σ is the frequency associated with the linear term.

In this function, the frequency σ is split in 2 parts: the dissipation at small scales and the dissipation at large scale (hypo-viscosity, *params.nu_m4*).

Returns **f_d** : *numpy.array*

The dissipation frequency as a function of the wavenumber (small sclale part).

f_d_hypo : *numpy.array*

The dissipation frequency at large scale (hypo-viscosity)

tendencies_nonlin(*variables=None*)
Compute the nonlinear tendencies.

This function has to be overridden in a child class.

Returns **tendencies_fft** : *fluidsim.base.setofvariables.SetOfVariables*

An array containing only zeros.

class fluidsim.base.solvers.pseudo_spect.**InfoSolverPseudoSpectral3D**(***kargs*)
Bases: *fluidsim.base.solvers.pseudo_spect.InfoSolverPseudoSpectral*

Contain the information on a base pseudo-spectral 3D solver.

_init_root()
Init. *self* by writting the information on the solver.

The first-level classes for this base solver are the same as for the 2D pseudo-spectral base solver except the class:

- *fluidsim.operators.operators.OperatorsPseudoSpectral2D*

Classes

InfoSolverPseudoSpectral(**kargs)	Contain the information on a base pseudo-spectral 2D solver.
InfoSolverPseudoSpectral3D(**kargs)	Contain the information on a base pseudo-spectral 3D solver.
Simul	alias of *SimulBasePseudoSpectral*
SimulBasePseudoSpectral(params)	Pseudo-spectral base solver.

fluidsim.base.solvers.finite_diff

Classes

InfoSolverFiniteDiff(**kargs)	Contain the information on a solver.

fluidsim.base.params

Information on a solver (fluidsim.base.params)

Provides:

class fluidsim.base.params.**Parameters**(*tag=None*, *attribs=None*, *path_file=None*, *elemxml=None*, *hdf5_object=None*)

Bases: fluiddyn.util.paramcontainer.ParamContainer

Contain the parameters.

Functions

create_params(input_info_solver)	Create a Parameters instance from an InfoSolverBase instance.
load_info_solver([path_dir])	Load the solver information, return an InfoSolverBase instance.
load_params_simul([path_dir])	Load the parameters and return a Parameters instance.

Classes

Parameters([tag, attribs, path_file, ...])	Contain the parameters.

fluidsim.base.setofvariables

Variable container (fluidsim.base.setofvariables)

Provides:

class fluidsim.base.setofvariables.**SetOfVariables**

Bases: numpy.ndarray

np.ndarray containing the variables.

Parameters **input_array** : `numpy.ndarray`, optional

Input array, default to None.

keys : Iterable, optional

Keys corresponding to the variables, default to None.

shape_variable : Iterable, optional

Shape of an array containing one variable, default to None.

like : *SetOfVariables*, optional

Model for creating the new *SetOfVariables*, default to None.

value : Number, optional

For initialization of the new *SetOfVariables*, default to None.

info : `str`, optional

Description, default to None.

dtype : `numpy.dtype`, optional

see `numpy.ndarray` help.

get_var(*arg*)
Get a variable as a np.array.

initialize(*value=0*)
Initialize as a constant array.

set_var(*arg*, *value*)
Set a variable.

Classes

SetOfVariables	np.ndarray containing the variables.

fluidsim.base.state

State of the variables (`fluidsim.base.state`)

Provides:

class `fluidsim.base.state.`**StateBase**(*sim*, *oper=None*)
Bases: `object`

Contains the state variables and handles the access to fields.

static **_complete_info_solver**(*info_solver*)
Complete the ParamContainer info_solver.

This is a static method!

class `fluidsim.base.state.`**StatePseudoSpectral**(*sim*, *oper=None*)
Bases: *fluidsim.base.state.StateBase*

Contains the state variables and handles the access to fields.

This is the general class for the pseudo-spectral solvers.

static _complete_info_solver(*info_solver*)
Complete the ParamContainer info_solver.

This is a static method!

init_statefft_from(***kwargs*)
Initialize *state_fft* from arrays.

Parameters ****kwargs** : {key: array, ...}

keys and arrays used for the initialization. The other keys are set to zero.

Examples

```
kwargs = {'a_fft': Fa_fft}
init_statefft_from(**kwargs)

ux_fft, uy_fft, eta_fft = oper.uxuyetafft_from_qfft(q_fft)
init_statefft_from(ux_fft=ux_fft, uy_fft=uy_fft, eta_fft=eta_fft)
```

return_statephys_from_statefft(*state_fft=None*)
Return the state in physical space.

Functions

Classes

StateBase(sim[, oper])	Contains the state variables and handles the access to fields.
StatePseudoSpectral(sim[, oper])	Contains the state variables and handles the access to fields.

fluidsim.base.init_fields

Initialisation of the fields (fluidsim.base.init_fields)

Provides:

class fluidsim.base.init_fields.**InitFieldsBase**(*sim*)
Bases: object

Initialization of the fields (base class).

static _complete_info_solver(*info_solver*, *classes=None*)
Complete the ParamContainer info_solver.

This is a static method!

static _complete_params_with_default(*params*, *info_solver*)
This static method is used to complete the *params* container.

Functions

Classes

InitFieldsBase(sim)	Initialization of the fields (base class).
InitFieldsConstant(sim)	
InitFieldsFromFile(sim)	
InitFieldsFromSimul(sim)	
InitFieldsManual(sim)	
SpecificInitFields(sim)	

fluidsim.base.time_stepping

Time stepping (fluidsim.base.time_stepping)

Provides:

base	Time stepping (*fluidsim.base.time_stepping.base*)
pseudo_spect	Time stepping (*fluidsim.base.time_stepping.pseudo_spect*)
pseudo_spect_cy	Time stepping Cython (*fluidsim.base.time_stepping.pseudo_spect_cy*)
finite_diff	Time stepping (*fluidsim.base.time_stepping.finite_diff*)

fluidsim.base.time_stepping.base

Time stepping (fluidsim.base.time_stepping.base) Provides:

class fluidsim.base.time_stepping.base.**TimeSteppingBase**(*sim*)

Bases: object

Universal time stepping class used for all solvers.

static _complete_params_with_default(*params*)

This static method is used to complete the *params* container.

_compute_time_increment_CLF_no_ux()

Compute the time increment deltat with a CLF condition.

_compute_time_increment_CLF_ux()

Compute the time increment deltat with a CLF condition.

_compute_time_increment_CLF_uxuy()

Compute the time increment deltat with a CLF condition.

_compute_time_increment_CLF_uxuyeta()

Compute the time increment deltat with a CLF condition.

_compute_time_increment_CLF_uxuyuz()

Compute the time increment deltat with a CLF condition.

one_time_step()

Main time stepping function.

start()

Loop to run the function *one_time_step()*.

If *self.USE_T_END* is true, run till `t >= t_end`, otherwise run *self.it_end* time steps.

Functions

Classes

TimeSteppingBase(sim)	Universal time stepping class used for all solvers.

fluidsim.base.time_stepping.pseudo_spect

Time stepping (`fluidsim.base.time_stepping.pseudo_spect`) Provides:

class `fluidsim.base.time_stepping.pseudo_spect.`**`TimeSteppingPseudoSpectral`**(*sim*)

Bases: *fluidsim.base.time_stepping.base.TimeSteppingBase*

Time stepping class for pseudo-spectral solvers.

`_time_step_RK2`()

Advance in time with the Runge-Kutta 2 method.

Notes

We consider an equation of the form

$$\partial_t S = \sigma S + N(S),$$

The Runge-Kutta 2 method computes an approximation of the solution after a time increment dt. We denote the initial time $t = 0$.

•Approximation 1:

$$\partial_t \log S = \sigma + \frac{N(S_0)}{S_0},$$

Integrating from t to $t + dt/2$, it gives:

$$S_{A1dt/2} = (S_0 + N_0 dt/2) e^{\frac{\sigma dt}{2}}.$$

•Approximation 2:

$$\partial_t \log S = \sigma + \frac{N(S_{A1dt/2})}{S_{A1dt/2}},$$

Integrating from t to $t + dt$ and retaining only the terms in dt^1 gives:

$$S_{dtA2} = S_0 e^{\sigma dt} + N(S_{A1dt/2}) dt e^{\frac{\sigma dt}{2}}.$$

`_time_step_RK4`()

Advance in time with the Runge-Kutta 4 method. We consider an equation of the form

$$\partial_t S = \sigma S + N(S),$$

The Runge-Kutta 4 method computes an approximation of the solution after a time increment dt. We denote the initial time as $t = 0$. This time scheme uses 4 approximations. Only the terms in dt^1 are retained.

•Approximation 1:

$$\partial_t \log S = \sigma + \frac{N(S_0)}{S_0},$$

Integrating from t to $t + dt/2$ gives:

$$S_{A1dt/2} = (S_0 + N_0 dt/2) e^{\sigma \frac{dt}{2}}.$$

Integrating from t to $t + dt$ gives:

$$S_{A1dt} = (S_0 + N_0 dt) e^{\sigma dt}.$$

•Approximation 2:

$$\partial_t \log S = \sigma + \frac{N(S_{A1dt/2})}{S_{A1dt/2}},$$

Integrating from t to $t + dt/2$ gives:

$$S_{A2dt/2} = S_0 e^{\sigma \frac{dt}{2}} + N(S_{A1dt/2}) \frac{dt}{2}.$$

Integrating from t to $t + dt$ gives:

$$S_{A2dt} = S_0 e^{\sigma dt} + N(S_{A1dt/2}) e^{\sigma \frac{dt}{2}} dt.$$

•Approximation 3:

$$\partial_t \log S = \sigma + \frac{N(S_{A2dt/2})}{S_{A2dt/2}},$$

Integrating from t to $t + dt$ gives:

$$S_{A3dt} = S_0 e^{\sigma dt} + N(S_{A2dt/2}) e^{\sigma \frac{dt}{2}} dt.$$

•Approximation 4:

$$\partial_t \log S = \sigma + \frac{N(S_{A3dt})}{S_{A3dt}},$$

Integrating from t to $t + dt$ gives:

$$S_{A4dt} = S_0 e^{\sigma dt} + N(S_{A3dt}) dt.$$

The final result is a pondered average of the results of 4 approximations for the time $t + dt$:

$$\frac{1}{3}\left[\frac{1}{2} S_{A1dt} + S_{A2dt} + S_{A3dt} + \frac{1}{2} S_{A4dt}\right],$$

which is equal to:

$$S_0 e^{\sigma dt} + \frac{dt}{3}\left[\frac{1}{2} N(S_0) e^{\sigma dt} + N(S_{A1dt/2}) e^{\sigma \frac{dt}{2}} + N(S_{A2dt/2}) e^{\sigma \frac{dt}{2}} + \frac{1}{2} N(S_{A3dt})\right].$$

one_time_step_computation()
One time step

Classes

ExactLinearCoefs(time_stepping)	Handle the computation of the exact coefficient for the RK4.
TimeSteppingPseudoSpectral(sim)	Time stepping class for pseudo-spectral solvers.

fluidsim.base.time_stepping.pseudo_spect_cy

Time stepping Cython (`fluidsim.base.time_stepping.pseudo_spect_cy`) Provides:

class fluidsim.base.time_stepping.pseudo_spect_cy.**ExactLinearCoefs**
Bases: fluidsim.base.time_stepping.pseudo_spect.ExactLinearCoefs

Handle the computation of the exact coefficient for the RK4.

class fluidsim.base.time_stepping.pseudo_spect_cy.**TimeSteppingPseudoSpectral**(*sim*)
Bases: *fluidsim.base.time_stepping.pseudo_spect.TimeSteppingPseudoSpectral*

_time_step_RK4_state_ndim3_freqlin_ndim2_float(*self*)
Advance in time *sim.state.state_fft* with the Runge-Kutta 4 method.

See *the pure python RK4 function* for the presentation of the time scheme.

For this function, the coefficient σ is real and represents the dissipation.

_time_step_RK4_state_ndim3_freqlin_ndim3_complex(*self*)
Advance in time *sim.state.state_fft* with the Runge-Kutta 4 method.

See *the pure python RK4 function* for the presentation of the time scheme.

For this function, the coefficient σ is complex.

_time_step_RK4_state_ndim3_freqlin_ndim3_float(*self*)
Advance in time *sim.state.state_fft* with the Runge-Kutta 4 method.

See *the pure python RK4 function* for the presentation of the time scheme.

For this function, the coefficient σ is complex.

_time_step_RK4_state_ndim4_freqlin_ndim3_float(*self*)
Advance in time *sim.state.state_fft* with the Runge-Kutta 4 method.

See *the pure python RK4 function* for the presentation of the time scheme.

For this function, the coefficient σ is real and represents the dissipation.

Functions

Classes

ExactLinearCoefs	Handle the computation of the exact coefficient for the RK4.
TimeSteppingPseudoSpectral(sim)	

fluidsim.base.time_stepping.finite_diff

Time stepping (`fluidsim.base.time_stepping.finite_diff`) Provides:

class fluidsim.base.time_stepping.finite_diff.**TimeSteppingFiniteDiffCrankNicolson**(*sim*)
Bases: *fluidsim.base.time_stepping.base.TimeSteppingBase*

Time stepping class for finite-difference solvers.

_time_step_RK2()
Advance in time the variables with the Runge-Kutta 2 method.

Notes

The Runge-Kutta 2 method computes an approximation of the solution after a time increment dt. We denote the initial time $t = 0$.

For the finite difference schemes, We consider an equation of the form

$$_tS = LS + N(S),$$

The linear term can be treated with an implicit method while the nonlinear term have to be treated with an explicit method (see for example Explicit and implicit methods).

•Approximation 1:

For the first step where the nonlinear term is approximated as $N(S) \simeq N(S_0)$, we obtain

$$\left(1 - \frac{dt}{4}L\right) S_{A1dt/2} \simeq \left(1 + \frac{dt}{4}L\right) S_0 + N(S_0)dt/2$$

Once the right-hand side has been computed, a linear equation has to be solved. It is not efficient to invert the matrix $1 + \frac{dt}{2}L$ so other methods have to be used, as the Thomas algorithm, or algorithms based on the LU or the QR decompositions.

•Approximation 2:

The nonlinear term is then approximated as $N(S) \simeq N(S_{A1dt/2})$, which gives

$$\left(1 - \frac{dt}{2}L\right) S_{A2dt} \simeq \left(1 + \frac{dt}{2}L\right) S_0 + N(S_{A1dt/2})dt$$

invert_to_get_solution(*A*, *b*)
Solve the linear system $Ax = b$.

one_time_step_computation()
One time step

Functions

Classes

TimeSteppingFiniteDiffCrankNicolson(sim)	Time stepping class for finite-difference solvers.

fluidsim.base.output

Output (fluidsim.base.output)

Provides:

base	Base module for the output (*fluidsim.base.output.base*)
prob_dens_func	
spectra	
phys_fields	Physical fields output (*fluidsim.base.output.phys_fields*)
spatial_means	
time_signalsK	
spatial_means	
time_signalsK	
increments	
print_stdout	
spect_energy_budget	

fluidsim.base.output.base

Base module for the output (fluidsim.base.output.base) Provides:

class fluidsim.base.output.base.**OutputBase**(*sim*)
Bases: object

Handle the output.

static _complete_info_solver(*info_solver*)
Complete the ParamContainer info_solver.

static _complete_params_with_default(*params*, *info_solver*)
This static method is used to complete the *params* container.

class fluidsim.base.output.base.**OutputBasePseudoSpectral**(*sim*)
Bases: *fluidsim.base.output.base.OutputBase*

class fluidsim.base.output.base.**SpecificOutput**(*output*, *period_save=0*, *period_plot=0*, *has_to_plot_saved=False*, *dico_arrays_1time=None*)
Bases: object

Small class for features useful for specific outputs

online_save()
Save the values at one time.

Functions

Classes

OutputBase(sim)	Handle the output.

Continued on next page

Table 2.25 – continued from previous page

OutputBasePseudoSpectral(sim)	
SpecificOutput(output[, period_save, ...])	Small class for features useful for specific outputs

fluidsim.base.output.prob_dens_func

Classes

ProbaDensityFunc(output)	Handle the saving and plotting of pdf of the turbulent kinetic energy.

fluidsim.base.output.spectra

Classes

Spectra(output)	Used for the saving of spectra.

fluidsim.base.output.phys_fields

Physical fields output (`fluidsim.base.output.phys_fields`) Provides:

class fluidsim.base.output.phys_fields.**PhysFieldsBase**(*output*)

Bases: *fluidsim.base.output.base.SpecificOutput*

Manage the output of physical fields.

online_plot()

Online plot.

online_save()

Online save.

Classes

PhysFieldsBase(output)	Manage the output of physical fields.
PhysFieldsBase1D(output)	
PhysFieldsBase2D(output)	

fluidsim.base.output.spatial_means

Functions

inner_prod(a_fft, b_fft)

Classes

SpatialMeansBase(output)	A SpatialMean object handles the saving of .

fluidsim.base.output.time_signalsK

Classes

TimeSignalsK(output)	A TimeSignalK object handles the saving of time signals in spectral space.

fluidsim.base.output.spatial_means

Functions

inner_prod(a_fft, b_fft)

Classes

SpatialMeansBase(output)	A SpatialMean object handles the saving of .

fluidsim.base.output.time_signalsK

Classes

TimeSignalsK(output)	A TimeSignalK object handles the saving of time signals in spectral space.

fluidsim.base.output.increments

Classes

Increments(output)	A Increments object handles the saving of pdf of increments.
IncrementsSW1L(output)	A Increments object handles the saving of pdf of increments.

fluidsim.base.output.print_stdout

Functions

Classes

PrintStdOutBase(output)	A PrintStdOutBase object is used to print in both the stdout and the stdout.txt file, and also to p

fluidsim.base.output.spect_energy_budget

Functions

cumsum_inv(a)
inner_prod(a_fft, b_fft)

Classes

SpectralEnergyBudgetBase(output)	Handle the saving and plotting of spectral energy budget.

class fluidsim.base.output.**OutputBase**(*sim*)
Bases: object

Handle the output.

static _complete_info_solver(*info_solver*)
Complete the ParamContainer info_solver.

static _complete_params_with_default(*params*, *info_solver*)
This static method is used to complete the *params* container.

class fluidsim.base.output.**OutputBasePseudoSpectral**(*sim*)
Bases: *fluidsim.base.output.base.OutputBase*

Functions

create_description_xmf_file([path])

Classes

fluidsim.base.forcing

Forcing schemes (fluidsim.base.forcing)

Provides:

base	Forcing schemes (*fluidsim.base.forcing.base*)
specific	Forcing schemes (*fluidsim.base.forcing.specific*)

fluidsim.base.forcing.base

Forcing schemes (`fluidsim.base.forcing.base`) Provides:

class `fluidsim.base.forcing.base.`**`ForcingBase`**(*sim*)
Bases: `object`

static `_complete_info_solver`(*info_solver*, *classes=None*)
Complete the ParamContainer info_solver.

static `_complete_params_with_default`(*params*, *info_solver*)
This static method is used to complete the *params* container.

class `fluidsim.base.forcing.base.`**`ForcingBasePseudoSpectral`**(*sim*)
Bases: *fluidsim.base.forcing.base.ForcingBase*

static `_complete_params_with_default`(*params*, *info_solver*)
This static method is used to complete the *params* container.

Classes

ForcingBase(sim)
ForcingBasePseudoSpectral(sim)

fluidsim.base.forcing.specific

Forcing schemes (`fluidsim.base.forcing.specific`) Provides:

class `fluidsim.base.forcing.specific.`**`SpecificForcing`**(*sim*)
Bases: `object`

class `fluidsim.base.forcing.specific.`**`SpecificForcingPseudoSpectral`**(*sim*)
Bases: *fluidsim.base.forcing.specific.SpecificForcing*

`compute`()
compute the forcing from a coarse forcing.

`put_forcingc_in_forcing`()
Copy data from forcingc_fft into forcing_fft.

`verify_injection_rate`()
Verify injection rate.

class `fluidsim.base.forcing.specific.`**`NormalizedForcing`**(*sim*)
Bases: *fluidsim.base.forcing.specific.SpecificForcingPseudoSpectral*

classmethod `_complete_params_with_default`(*params*)
This static method is used to complete the *params* container.

`coef_normalization_from_abc`(*a*, *b*, *c*)
.

`compute`()
compute a forcing normalize with a 2nd degree eq.

`normalize_forcingc_2nd_degree_eq`(*fvc_fft*, *vc_fft*)
Modify the array fvc_fft to fixe the injection rate.

To be called only with proc 0.

Parameters **fvc_fft** : ndarray

The non-normalized forcing at the coarse resolution.

vc_fft : ndarray

The forced variable at the coarse resolution.

normalize_forcingc_part_k(*fvc_fft*, *vc_fft*)
Modify the array fvc_fft to fixe the injection rate.

To be called only with proc 0.

Parameters **fvc_fft** : ndarray

The non-normalized forcing at the coarse resolution.

vc_fft : ndarray

The forced variable at the coarse resolution.

class fluidsim.base.forcing.specific.**Proportional**(*sim*)
Bases: *fluidsim.base.forcing.specific.SpecificForcingPseudoSpectral*

normalize_forcingc(*vc_fft*)
Modify the array fvc_fft to fixe the injection rate.

varc [ndarray] a variable at the coarse resolution.

To be called only with proc 0.

class fluidsim.base.forcing.specific.**RamdomSimplePseudoSpectral**(*sim*)
Bases: *fluidsim.base.forcing.specific.NormalizedForcing*

compute_forcingc_raw()
Random coarse forcing.

To be called only with proc 0.

class fluidsim.base.forcing.specific.**TimeCorrelatedRandomPseudoSpectral**(*sim*)
Bases: *fluidsim.base.forcing.specific.RamdomSimplePseudoSpectral*

classmethod **_complete_params_with_default**(*params*)
This static method is used to complete the *params* container.

Functions

Classes

NormalizedForcing(sim)
Proportional(sim)
RamdomSimplePseudoSpectral(sim)
SpecificForcing(sim)
SpecificForcingPseudoSpectral(sim)
TimeCorrelatedRandomPseudoSpectral(sim)

Classes

fluidsim.base.preprocess

Preprocessing of parameters (`fluidsim.base.preprocess`)

Provides:

base	Base preprocess (`fluiddyn.simul.base.preprocess.base`)
pseudo_spect	Preprocessing for pseudo-spectral solvers (`fluiddyn.simul.base.preprocess.pseudo_spect`)

fluidsim.base.preprocess.base

Base preprocess (`fluiddyn.simul.base.preprocess.base`) Provides:

class `fluidsim.base.preprocess.base.`**`PreprocessBase`**(*sim*)
Bases: `object`

static `_complete_info_solver`(*info_solver*, *classes=None*)
Complete the ParamContainer info_solver.

static `_complete_params_with_default`(*params*, *info_solver*)
This static method is used to complete the *params* container.

Classes

PreprocessBase(sim)

fluidsim.base.preprocess.pseudo_spect

Preprocessing for pseudo-spectral solvers (`fluiddyn.simul.base.preprocess.pseudo_spect`) Provides:

class `fluidsim.base.preprocess.pseudo_spect.`**`PreprocessPseudoSpectral`**(*sim*)
Bases: *fluidsim.base.preprocess.base.PreprocessBase*

`normalize_init_fields`()
A non-dimensionalization procedure for the initialized fields.

`set_forcing_rate`()
Based on C, a non-dimensional ratio of forcing rate to one of the following forcing scales

•the initial total energy, math:: E_0

•the initial total enstrophy, math:: Omega_0

the forcing rate is set.

Parameters **params.preprocess.forcing_const** : float

Non-dimensional ratio of forcing_scale to forcing_rate

params.preprocess.forcing_scale : string

Mean quantity to be scaled against

.. **TODO** : Trivial error in computing forcing rate

set_viscosity()

Based on

•the initial total enstrophy, Omega_0, or

•the initial energy

•the forcing rate, epsilon

the viscosity scale or Reynolds number is set.

Parameters **params.preprocess.viscosity_type** : string

Type/Order of viscosity desired

params.preprocess.viscosity_scale : string

Mean quantity to be scaled against

params.preprocess.viscosity_const : float

Calibration constant to set dissipative wave number

Classes

PreprocessPseudoSpectral(sim)	

2.2 fluidsim.operators

2.2.1 Numerical operators

Provides

operators	Numerical operators (*fluidsim.operators.operators*)
fft	Fast Fourier transforms (*fluidsim.operators.fft*)

fluidsim.operators.operators

Numerical operators (fluidsim.operators.operators)

This module is written in Cython and provides the classes:

class fluidsim.operators.operators.**Operators**

Bases: object

class fluidsim.operators.operators.**GridPseudoSpectral2D**

Bases: *fluidsim.operators.operators.Operators*

Describes a discretisation in spectral and spatial space.

Parameters **nx, ny** : int

Number of colocation points in the x and y directions

Lx, Lx : float

Dimension of the numerical box in the x and y directions

op_fft2d : `FFT2Dmpi`

A instance of the class `OP_FFT2Dmpi`

SEQUENTIAL : bool

If True, the fft is sequential even though `nb_proc > 1`

class `fluidsim.operators.operators.`**OperatorsPseudoSpectral2D**
Bases: *fluidsim.operators.operators.GridPseudoSpectral2D*

Provides fast Fourier transform functions and 2D operators.

type_fft='FFTWCY' : cython wrapper of plans fftw_plan_dft_r2c_2d / fftw_plan_dft_c2r_2d (sequential case) and fftw_mpi_plan_dft_r2c_2d / fftw_mpi_plan_dft_c2r_2d (parallel case)

type_fft='FFTWCCY' : cython wrapper of a self-written c libray using sequential fftw plans and MPI_Type. Seems to be faster than the implementation of the mpi FFT by fftw (lib fftw-mpi).

type_fft='FFTWPY' : use of the module `easypyfft2D` with fftw

type_fft='FFTPACK' use of the module :mod:'easypyfft2D with fftp (bad and slow implementation!)

static **_complete_params_with_default**()
This static method is used to complete the *params* container.

apamfft_from_adfft()
Return the eigen modes ap and am.

coarse_seq_from_fft_loc()
Return a coarse field in K space.

compute_increments_dim1()
Compute the increments of var over the dim 1.

constant_arrayK()
Return a constant array in spectral space.

constant_arrayX()
Return a constant array in real space.

divfft_from_apamfft()
Return div from the eigen modes ap and am.

divfft_from_vecfft()
Return the divergence of a vector in spectral space.

etafft_from_afft()
Compute eta in Fourier space.

etafft_from_aqfft()
Compute eta in Fourier space.

etafft_from_qfft()
Compute eta in Fourier space.

fft_loc_from_coarse_seq()
Return a large field in K space.

gradfft_from_fft()
Return the gradient of f_fft in spectral space.

pdf_normalized()
Compute the normalized pdf

produce_long_str_describing_oper()
Produce a string describing the operator.

produce_str_describing_oper()
Produce a string describing the operator.

project_fft_on_realX_seq()
Project the given field in spectral space such as its inverse fft is a real field.

pxfft_from_fft()
Return the gradient of f_fft in spectral space.

pyfft_from_fft()
Return the gradient of f_fft in spectral space.

qapamfft_from_uxuyetafft()
ux, uy, eta (fft) —> q, ap, am (fft)

qapamfft_from_uxuyetafft_old()
ux, uy, eta (fft) —> q, ap, am (fft)

random_arrayK()
Return a random array in spectral space.

random_arrayX()
Return a random array in real space.

rotfft_from_afft()
Compute ux, uy and eta in Fourier space.

rotfft_from_qfft()
Compute ux, uy and eta in Fourier space.

rotfft_from_vecfft()
Return the rotational of a vector in spectral space.

spectra1D_from_fft()
Compute the 1D spectra. Return a dictionary.

spectrum2D_from_fft()
Compute the 2D spectra. Return a dictionary.

sum_wavenumbers()
Sum the given array over all wavenumbers.

uxuyetafft_from_afft()
Compute ux, uy and eta in Fourier space.

uxuyetafft_from_qapamfft()
q, ap, am (fft) —> ux, uy, eta (fft)

uxuyetafft_from_qapamfft_old()
q, ap, am (fft) —> ux, uy, eta (fft)

uxuyetafft_from_qfft()
Compute ux, uy and eta in Fourier space.

vecfft_from_divfft()
Return the velocity in spectral space computed from the divergence.

vecfft_from_rotfft()
Return the velocity in spectral space computed from the rotational.

Functions

Classes

GridPseudoSpectral2D	Describes a discretisation in spectral and spatial space.
Operators	
OperatorsPseudoSpectral2D	Provides fast Fourier transform functions and 2D operators.

fluidsim.operators.fft

Fast Fourier transforms (`fluidsim.operators.fft`)

Provides

easypyfft	Fast Fourier transforms (*fluidsim.operators.fft.easypyfft*)

fluidsim.operators.fft.easypyfft

Fast Fourier transforms (`fluidsim.operators.fft.easypyfft`) Provides classes for performing fft in 1, 2, and 3 dimensions:

class `fluidsim.operators.fft.easypyfft.`**`fftp2D`**(*nx*, *ny*)
A class to use fftp

class `fluidsim.operators.fft.easypyfft.`**`FFTW2DReal2Complex`**(*nx*, *ny*)
A class to use fftw

class `fluidsim.operators.fft.easypyfft.`**`FFTW3DReal2Complex`**(*nx*, *ny*, *nz*)
A class to use fftw

class `fluidsim.operators.fft.easypyfft.`**`FFTW1D`**(*n*)
A class to use fftw 1D

class `fluidsim.operators.fft.easypyfft.`**`FFTW1DReal2Complex`**(*arg*, *axis=-1*)
A class to use fftw 1D

Functions

Classes

FFTW1D(n)	A class to use fftw 1D
FFTW1DReal2Complex(arg[, axis])	A class to use fftw 1D
FFTW2DReal2Complex(nx, ny)	A class to use fftw
FFTW3DReal2Complex(nx, ny, nz)	A class to use fftw
fftp2D(nx, ny)	A class to use fftp

Other extension modules (fftw2dmpiccy, fftw2dmpicy) can not be imported on the readthedocs servers to can be be included here.

Todo

Make a nice hierarchy of Operators classes (?)

- Operators
- OperatorsPseudoSpectral1D
- OperatorsPseudoSpectral2D
- OperatorsPseudoSpectral3D
- OperatorsFiniteDiff1D
- OperatorsFiniteDiff2D
- OperatorsFiniteDiff3D
- OperatorsPseudoSpectral1DFiniteDiff1D
- OperatorsPseudoSpectral2DFiniteDiff1D

2.3 fluidsim.solvers

2.3.1 Particular solvers

Base package containing the source code of the particular solvers. It provides the following solver packages:

ns2d	Navier-Stokes 2D solvers (*fluidsim.solvers.ns2d*)
ns3d	Navier-Stokes 3D solvers (*fluidsim.solvers.ns3d*)
sw1l	1-layer Shallow-Water solvers (*fluidsim.solvers.sw1l*)
plate2d	Plate2d solvers (*fluidsim.solvers.plate2d*)
ad1d	Advection-diffusion 1D solvers (*fluidsim.solvers.ad1d*)
waves2d	2D waves solvers (*fluidsim.solvers.waves2d*)

fluidsim.solvers.ns2d

Navier-Stokes 2D solvers (`fluidsim.solvers.ns2d`)

Base package for the Navier-Stokes 2D solvers. The main solver defined in *fluidsim.solvers.ns2d.solver* is pseudo-spectral.

Provides:

solver	NS2D solver (*fluidsim.solvers.ns2d.solver*)
state	State for the NS2D solver (*fluidsim.solvers.ns2d.state*)
init_fields	Initialization of the field (*fluidsim.solvers.ns2d.init_fields*)
output	Output (*fluidsim.solvers.ns2d.output*)
forcing	Forcing (*fluidsim.solvers.ns2d.forcing*)

fluidsim.solvers.ns2d.solver

NS2D solver (`fluidsim.solvers.ns2d.solver`) This module provides two classes defining the pseudo-spectral solver 2D incompressible Navier-Stokes equations (ns2d).

class `fluidsim.solvers.ns2d.solver.`**`InfoSolverNS2D`**(***kargs*)
Bases: *fluidsim.base.solvers.pseudo_spect.InfoSolverPseudoSpectral*

Contain the information on the solver ns2d.

`_init_root`()
Init. *self* by writting the information on the solver.

The function *InfoSolverPseudoSpectral._init_root* is called. We keep two classes listed by this function:

- *fluidsim.base.time_stepping.pseudo_spect_cy.TimeSteppingPseudoSpectral*
- *fluidsim.operators.operators.OperatorsPseudoSpectral2D*

The other first-level classes for this solver are:

- *fluidsim.solvers.ns2d.solver.Simul*
- *fluidsim.solvers.ns2d.state.StateNS2D*
- *fluidsim.solvers.ns2d.init_fields.InitFieldsNS2D*
- *fluidsim.solvers.ns2d.output.Output*
- *fluidsim.solvers.ns2d.forcing.ForcingNS2D*

class `fluidsim.solvers.ns2d.solver.`**`Simul`**(*params*)
Bases: *fluidsim.base.solvers.pseudo_spect.SimulBasePseudoSpectral*

Pseudo-spectral solver 2D incompressible Navier-Stokes equations.

`InfoSolver`
alias of *InfoSolverNS2D*

static `_complete_params_with_default`(*params*)
Complete the *params* container (static method).

`tendencies_nonlin`(*state_fft=None*)
Compute the nonlinear tendencies.

Parameters **state_fft** : *fluidsim.base.setofvariables.SetOfVariables*

optional

Array containing the state, i.e. the vorticity, in Fourier space. If *state_fft*, the variables vorticity and the velocity are computed from it, otherwise, they are taken from the global state of the simulation, *self.state*.

These two possibilities are used during the Runge-Kutta time-stepping.

Returns **tendencies_fft** : *fluidsim.base.setofvariables.SetOfVariables*

An array containing the tendencies for the vorticity.

Notes

The 2D Navier-Stockes equation can be written

$$\partial_t \hat{\zeta} = \hat{N}(\zeta) - \sigma(k)\zeta,$$

This function compute the nonlinear term ("tendencies") $\hat{N}(\zeta) = -\mathbf{u} \cdot \nabla \zeta$.

Classes

InfoSolverNS2D(**kargs)	Contain the information on the solver ns2d.
Simul(params)	Pseudo-spectral solver 2D incompressible Navier-Stokes equations.

fluidsim.solvers.ns2d.state

State for the NS2D solver (`fluidsim.solvers.ns2d.state`)

class fluidsim.solvers.ns2d.state.**StateNS2D**(*sim*, *oper=None*)

Bases: *fluidsim.base.state.StatePseudoSpectral*

State for the solver ns2d.

Contains the variables corresponding to the state and handles the access to other fields.

static _complete_info_solver(*info_solver*)

Complete the *info_solver* container (static method).

compute(*key*, *SAVE_IN_DICT=True*, *RAISE_ERROR=True*)

Compute and return a variable

init_from_rotfft(*rot_fft*)

Initialize the state from the variable *rot_fft*.

statefft_from_statephys()

Compute *state_fft* from *state_phys*.

statephys_from_statefft()

Compute *state_phys* from *statefft*.

Classes

StateNS2D(sim[, oper])	State for the solver ns2d.

fluidsim.solvers.ns2d.init_fields

Initialization of the field (`fluidsim.solvers.ns2d.init_fields`)

class fluidsim.solvers.ns2d.init_fields.**InitFieldsNS2D**(*sim*)

Bases: *fluidsim.base.init_fields.InitFieldsBase*

Initialize the state for the solver NS2D.

class fluidsim.solvers.ns2d.init_fields.**InitFieldsNoise**(*sim*)

Bases: fluidsim.base.init_fields.SpecificInitFields

Initialize the state with noise.

class fluidsim.solvers.ns2d.init_fields.**InitFieldsJet**(*sim*)

Bases: fluidsim.base.init_fields.SpecificInitFields

Initialize the state with a jet.

class fluidsim.solvers.ns2d.init_fields.**InitFieldsDipole**(*sim*)

Bases: fluidsim.base.init_fields.SpecificInitFields

Initialize the state with a dipole.

Classes

InitFieldsDipole(sim)	Initialize the state with a dipole.
InitFieldsJet(sim)	Initialize the state with a jet.
InitFieldsNS2D(sim)	Initialize the state for the solver NS2D.
InitFieldsNoise(sim)	Initialize the state with noise.

fluidsim.solvers.ns2d.output

Output (`fluidsim.solvers.ns2d.output`) Provides the modules:

print_stdout	Simple text output (*fluidsim.solvers.ns2d.output.print_stdout*)
spatial_means	Spatial means output (*fluidsim.solvers.ns2d.output.spatial_means*)
spectra	Spectra output (*fluidsim.solvers.ns2d.output.spectra*)
spect_energy_budget	Energy budget (*fluidsim.solvers.ns2d.output.spect_energy_budget*)

fluidsim.solvers.ns2d.output.print_stdout

Simple text output (`fluidsim.solvers.ns2d.output.print_stdout`)

class `fluidsim.solvers.ns2d.output.print_stdout.`**`PrintStdOutNS2D`**(*output*)

Bases: `fluidsim.base.output.print_stdout.PrintStdOutBase`

Simple text output.

Used to print in both the stdout and the stdout.txt file, and also to print simple info on the current state of the simulation.

Classes

PrintStdOutNS2D(output)	Simple text output.

fluidsim.solvers.ns2d.output.spatial_means

Spatial means output (`fluidsim.solvers.ns2d.output.spatial_means`)

class `fluidsim.solvers.ns2d.output.spatial_means.`**`SpatialMeansNS2D`**(*output*)

Bases: `fluidsim.base.output.spatial_means.SpatialMeansBase`

Spatial means output.

Classes

SpatialMeansNS2D(output)	Spatial means output.

fluidsim.solvers.ns2d.output.spectra

Spectra output (`fluidsim.solvers.ns2d.output.spectra`)

class fluidsim.solvers.ns2d.output.spectra.**SpectraNS2D**(*output*)
Bases: fluidsim.base.output.spectra.Spectra

Save and plot spectra.

compute()
compute the values at one time.

Classes

SpectraNS2D(output)	Save and plot spectra.

fluidsim.solvers.ns2d.output.spect_energy_budget

Energy budget (fluidsim.solvers.ns2d.output.spect_energy_budget)
class fluidsim.solvers.ns2d.output.spect_energy_budget.**SpectralEnergyBudgetNS2D**(*output*)
Bases: fluidsim.base.output.spect_energy_budget.SpectralEnergyBudgetBase

Save and plot energy budget in spectral space.

compute()
compute the spectral energy budget at one time.

Functions

Classes

SpectralEnergyBudgetNS2D(output)	Save and plot energy budget in spectral space.

and the main output class for the ns2d solver:

class fluidsim.solvers.ns2d.output.**Output**(*sim*)
Bases: *fluidsim.base.output.base.OutputBasePseudoSpectral*

Output for ns2d solver.

static _complete_info_solver(*info_solver*)
Complete the *info_solver* container (static method).

static _complete_params_with_default(*params*, *info_solver*)
Complete the *params* container (static method).

compute_energy()
Compute the spatially averaged energy.

compute_energy_fft()
Compute energy(k)

compute_enstrophy()
Compute the spatially averaged enstrophy.

compute_enstrophy_fft()
Compute enstrophy(k)

Classes

Output(sim)	Output for ns2d solver.

fluidsim.solvers.ns2d.forcing

Forcing (fluidsim.solvers.ns2d.forcing)

class fluidsim.solvers.ns2d.forcing.**ForcingNS2D**(*sim*)
Bases: *fluidsim.base.forcing.base.ForcingBasePseudoSpectral*

Forcing class for the ns2d solver.

Classes

ForcingNS2D(sim)	Forcing class for the ns2d solver.

fluidsim.solvers.ns3d

Navier-Stokes 3D solvers (fluidsim.solvers.ns3d)

Provides:

solver	NS3D solver (*fluidsim.solvers.ns3d.solver*)

fluidsim.solvers.ns3d.solver

NS3D solver (fluidsim.solvers.ns3d.solver)

class fluidsim.solvers.ns3d.solver.**Simul**(*params*)
Bases: *fluidsim.base.solvers.pseudo_spect.SimulBasePseudoSpectral*

Pseudo-spectral solver 3D incompressible Navier-Stokes equations.

Not yet implemented!

Notes

This class is dedicated to solve with a pseudo-spectral method the incompressible Navier-Stokes equations (possibly with hyper-viscosity):

$$\partial_t \mathbf{v} + \mathbf{v} \cdot \boldsymbol{\nabla} \mathbf{v} = -\boldsymbol{\nabla} p - \nu_\alpha (-\Delta)^\alpha \mathbf{v},$$

where $\mathbf{v}$ is the non-divergent velocity ($\boldsymbol{\nabla} \cdot \mathbf{v} = 0$), p is the pressure, Δ is the 3D Laplacian operator.

In Fourier space, these equations can be written as:

$$\partial_t \hat{v} = N(v) + L\hat{v},$$

where

$$N(\mathbf{v}) = -P_\perp \widehat{\mathbf{v} \cdot \boldsymbol{\nabla} \mathbf{v}},$$

$$L = -\nu_\alpha |\mathbf{k}|^{2\alpha},$$

with $P_\perp = (1 - \hat{\mathbf{e}}_\mathbf{k} \hat{\mathbf{e}}_\mathbf{k} \cdot)$ the operator projection on the plane perpendicular to the wave number $\mathbf{k}$.

static _complete_params_with_default(*params*)
This static method is used to complete the *params* container.

Todo

- 3D pseudo-spectral operator with parallel fft,
- output and 3D plotting,

Classes

InfoSolverNS3D(**kargs)	
Simul(params)	Pseudo-spectral solver 3D incompressible Navier-Stokes equations.

fluidsim.solvers.sw1l

1-layer Shallow-Water solvers (fluidsim.solvers.sw1l)

fluidsim.solvers.plate2d

Plate2d solvers (fluidsim.solvers.plate2d)

Provides:

solver	Plate2d solver (*fluidsim.solvers.plate2d.solver*)
state	Plate2d state (*fluidsim.solvers.plate2d.state*)
output	Plate2d output (*fluidsim.solvers.plate2d.output*)
init_fields	Plate2d InitFields (*fluidsim.solvers.plate2d.init_fields*)
forcing	Plate2d forcing (*fluidsim.solvers.plate2d.forcing*)
diag	Plate2d solvers diagonalized (*fluidsim.solvers.plate2d.diag*)
dimensional	Plate2d solver (*fluidsim.solvers.plate2d.dimensional*)

fluidsim.solvers.plate2d.solver

Plate2d solver (fluidsim.solvers.plate2d.solver) This module provides two classes defining the pseudo-spectral solver plate2d.

class fluidsim.solvers.plate2d.solver.**InfoSolverPseudoSpectral**(**kargs*)
Bases: fluidsim.base.solvers.info_base.InfoSolverBase

Contain the information on a base pseudo-spectral 2D solver.

_init_root()
Init. *self* by writting the information on the solver.

The first-level classes for this base solver are:

- *fluidsim.base.solvers.pseudo_spect.SimulBasePseudoSpectral*
- *fluidsim.base.state.StatePseudoSpectral*
- *fluidsim.base.time_stepping.pseudo_spect_cy.TimeSteppingPseudoSpectral*
- *fluidsim.operators.operators.OperatorsPseudoSpectral2D*

class fluidsim.solvers.plate2d.solver.**Simul**(*params*)
Bases: *fluidsim.base.solvers.pseudo_spect.SimulBasePseudoSpectral*

Pseudo-spectral solver solving the Föppl-von Kármán equations.

Notes

This class is dedicated to solve with a pseudo-spectral method the Föppl-von Kármán equations which describe the dynamics of a rigid plate. Using the non-dimensional variables displacement z and out of plane velocity w:

$$\partial_t z = w,$$

$$\partial_t w = -\Delta^2 z + N_w(z) + f_w - \nu_\alpha(-\Delta)^\alpha w.$$

where $\Delta = \partial_{xx} + \partial_{yy}$ is the Laplacian. The first term of the two equations corresponds to the linear part. f_w and $\nu_\alpha \Delta^\alpha w$ are the forcing and the dissipation terms, respectively. The nonlinear term is equal to $N_w(z) = \{z, \chi\}$, where $\{\cdot, \cdot\}$ is the Monge-Ampère operator

$$\{a, b\} = \partial_{xx} a \partial_{yy} b + \partial_{yy} a \partial_{xx} b - 2\partial_{xy} a \partial_{xy} b,$$

and

$$\Delta^2 \chi = -\{z, z\}.$$

Taking the Fourier transform, we get:

$$\partial_t \hat{z} = \hat{w},$$

$$\partial_t \hat{w} = -k^4 \hat{z} + \widehat{N_w(z)} + \hat{f}_w - \nu_\alpha k^{2\alpha} \hat{w},$$

where $k^2 = |\mathbf{k}|^2$. For this simple solver, we will use the variables z and w and only the dissipative term will be solve exactly. Thus, all the other terms are included in the *tendencies_nonlin()* function.

Energetics: The total energy can be decomposed in the kinetic energy

$$E_K = \frac{1}{2}\langle w^2 \rangle = \frac{1}{2} \sum_{\mathbf{k}} |\hat{w}|^2,$$

the flexion energy

$$E_L = \frac{1}{2}\langle (\Delta z)^2 \rangle = \frac{1}{2} \sum_{\mathbf{k}} k^4 |\hat{z}|^2,$$

and the non-quadratic extension energy

$$E_E = \frac{1}{4}\langle (\Delta \chi)^2 \rangle = \frac{1}{4} \sum_{\mathbf{k}} k^4 |\hat{\chi}|^2.$$

The energy injected into the system by the forcing is

$$P = \langle f_w w \rangle,$$

and the dissipation is

$$D = \nu_\alpha \langle w(-\Delta)^\alpha w \rangle.$$

InfoSolver
alias of `InfoSolverPlate2D`

static _complete_params_with_default(*params*)
Complete the *params* container (static method).

compute_freq_diss()
Compute the dissipation frequencies with dissipation only for w.

tendencies_nonlin(*state_fft=None*)
Compute the "nonlinear" tendencies.

test_tendencies_nonlin(*tendencies_fft=None*, *w_fft=None*, *z_fft=None*, *chi_fft=None*)
Test if the tendencies conserves the total energy.

We consider the conservative Föppl-von Kármán equations (without dissipation and forcing) written as

$$\partial_t z = F_z$$
$$\partial_t w = F_w$$

We have:

$$\partial_t E_K(\mathbf{k}) = \mathcal{R}(\hat{F}_w \hat{w}^*)$$
$$\partial_t E_L(\mathbf{k}) = k^4 \mathcal{R}(\hat{F}_z \hat{z}^*)$$
$$\partial_t E_{NQ}(\mathbf{k}) = -\mathcal{R}(\{\widehat{F_z, z}\}\hat{\chi}^*)$$

Since the total energy is conserved, we should have

$$\sum_{\mathbf{k}} \partial_t E_K(\mathbf{k}) + \partial_t E_L(\mathbf{k}) + \partial_t E_{NQ}(\mathbf{k}) = 0$$

This function computes this quantities.

Todo

- other solver solving exactly the linear terms.

Classes

`InfoSolverPlate2D`(**kargs)	Contain the information on the solver plate2d.
Simul(params)	Pseudo-spectral solver solving the Föppl-von Kármán equations.

fluidsim.solvers.plate2d.state

Plate2d state (`fluidsim.solvers.plate2d.state`)

Classes

StatePlate2D(sim[, oper])	Contains the variables corresponding to the state and handles the access to other fields for the solver

fluidsim.solvers.plate2d.output

Plate2d output (`fluidsim.solvers.plate2d.output`) Provides:

print_stdout	Standard output (*fluidsim.solvers.plate2d.output.print_stdout*)
spatial_means	Spatial means (*fluidsim.solvers.plate2d.output.spatial_means*)
spectra	Spectra (*fluidsim.solvers.plate2d.output.spectra*)
correlations_freq	Correl freq (*fluidsim.solvers.plate2d.output.correlations_freq*)

fluidsim.solvers.plate2d.output.print_stdout

Standard output (`fluidsim.solvers.plate2d.output.print_stdout`) Provides:

class fluidsim.solvers.plate2d.output.print_stdout.**PrintStdOutPlate2D**(*output*)
Bases: *fluidsim.solvers.ns2d.output.print_stdout.PrintStdOutNS2D*

Used to print in both the stdout and the stdout.txt file, and also to print simple info on the current state of the simulation.

Classes

PrintStdOutPlate2D(output)	Used to print in both the stdout and the stdout.txt file, and also to print simple info on the curren

fluidsim.solvers.plate2d.output.spatial_means

Spatial means (`fluidsim.solvers.plate2d.output.spatial_means`) Provides:

class fluidsim.solvers.plate2d.output.spatial_means.**SpatialMeansPlate2D**(*output*)
Bases: fluidsim.base.output.spatial_means.SpatialMeansBase

Compute, save, load and plot spatial means.

If only W is forced and dissipated, the energy budget is quite simple and can be written as:

$$\partial_t E_w = -C_{w\to z} - C_{w\to\chi} + P_w - D_w,$$
$$\partial_t E_z = +C_{w\to z},$$
$$\partial_t E_\chi = +C_{w\to\chi},$$

where

$$C_{w\to z} = \sum_{\mathbf{k}} k^4 \mathcal{R}(\hat{w}\hat{z}^*),$$
$$C_{w\to\chi} = -\sum_{\mathbf{k}} \mathcal{R}(\{\widehat{w,z}\}\hat{\chi}^*),$$
$$P_w = \sum_{\mathbf{k}} \mathcal{R}(\hat{f}_w\hat{w}^*)$$

and

$$D_w = 2\nu_\alpha \sum_{\mathbf{k}} k^{2\alpha} E_w(k).$$

Classes

SpatialMeansPlate2D(output)	Compute, save, load and plot spatial means.

fluidsim.solvers.plate2d.output.spectra

Spectra (`fluidsim.solvers.plate2d.output.spectra`) Provides:

class `fluidsim.solvers.plate2d.output.spectra.`**`SpectraPlate2D`**(*output*)
Bases: `fluidsim.base.output.spectra.Spectra`

Compute, save, load and plot spectra.

`compute`()
compute the values at one time.

Classes

SpectraPlate2D(output)	Compute, save, load and plot spectra.

fluidsim.solvers.plate2d.output.correlations_freq

Correl freq (`fluidsim.solvers.plate2d.output.correlations_freq`) Provides:

class `fluidsim.solvers.plate2d.output.correlations_freq.`**`CorrelationsFreq`**(*output*)
Bases: *fluidsim.base.output.base.SpecificOutput*

Compute, save, load and plot correlations of frequencies.

`_compute_correl2`(*q_fftt*)
Compute the correlations 2.

$$C_2(\omega_1, \omega_2) = \langle \tilde{w}(\omega_1, \mathbf{x}) \tilde{w}(\omega_2, \mathbf{x})^* \rangle_{\mathbf{x}}.$$

`_compute_correl4`(*q_fftt*)
Compute the correlations 4.

$$C_4(\omega_1, \omega_2, \omega_3, \omega_4) = \langle \tilde{w}(\omega_1, \mathbf{x}) \tilde{w}(\omega_2, \mathbf{x}) \tilde{w}(\omega_3, \mathbf{x})^* \tilde{w}(\omega_4, \mathbf{x})^* \rangle_{\mathbf{x}},$$

where

$$\omega_2 = \omega_3 + \omega_4 - \omega_1$$

and $\omega_1 > 0$, $\omega_3 > 0$ and

$\omega_4 > 0$. Thus, this function produces an array $C_4(\omega_1, \omega_3, \omega_4)$.

`online_save`()
Save the values at one time.

Functions

Classes

CorrelationsFreq(output)	Compute, save, load and plot correlations of frequencies.

Classes

Output(sim)

fluidsim.solvers.plate2d.init_fields

Plate2d InitFields (fluidsim.solvers.plate2d.init_fields)

Classes

InitFieldsHarmonic(sim)	
InitFieldsNoise(sim)	
InitFieldsPlate2D(sim)	Init the fields for the solver PLATE2D.

fluidsim.solvers.plate2d.forcing

Plate2d forcing (fluidsim.solvers.plate2d.forcing)

Classes

ForcingPlate2D(sim)
Proportional(sim)
TCRandomPSW(sim)
TimeCorrelatedRandomPseudoSpectral(sim)

fluidsim.solvers.plate2d.diag

Plate2d solvers diagonalized (fluidsim.solvers.plate2d.diag) Provides:

solver	Plate2d solver diag.

fluidsim.solvers.plate2d.diag.solver

Plate2d solver diag. (fluidsim.solvers.plate2d.diag.solver) Provides:

class fluidsim.solvers.plate2d.diag.solver.**Simul**(*params*)

Bases: *fluidsim.base.solvers.pseudo_spect.SimulBasePseudoSpectral*

Pseudo-spectral solver solving the Föppl-von Kármán equations.

Notes

This class is dedicated to solve with a pseudo-spectral method the Föppl-von Kármán equations which describe the dynamics of a rigid plate (see *fluidsim.solvers.plate2d.solver.Simul*).

In the Fourier space, the governing equations write:

$$\partial_t \hat{z} = \hat{w},$$
$$\partial_t \hat{w} = -\Omega(k)^2 \hat{z} + \widehat{N_w(z)} + \hat{F}_w - \gamma_w \hat{w},$$

where $\Omega(k) = k^4$, $k^2 = |\mathbf{k}|^2$ and $\gamma_w = \nu_\alpha k^{2\alpha}$. For this solver, we will use variables that diagonalized the linear terms, i.e. that represent propagative waves (see *fluidsim.solvers.waves2d.solver.Simul*). Therefore, all the linear terms can be solved exactly. Applying the equations of *fluidsim.solvers.waves2d.solver.Simul* with $\gamma_f = \gamma_z = 0$, we find that the eigenvalues are $\sigma_\pm = -\gamma_w/2 \pm i\tilde{\Omega}$ with $\tilde{\Omega} = \Omega\sqrt{1-(\gamma_w/(2\Omega))^2}$, and the (not normalized) eigenvectors are

$$V_\pm = \begin{pmatrix} 1 \\ \sigma_\pm \end{pmatrix}.$$

The state can be represented by a vector A that verifies $X = VA$, where V is the base matrix

$$V = \begin{pmatrix} 1 & 1 \\ \sigma_+ & \sigma_- \end{pmatrix}.$$

The inverse base matrix is given by

$$V^{-1} = \frac{i}{2\tilde{\Omega}} \begin{pmatrix} \sigma_- & -1 \\ -\sigma_+ & 1 \end{pmatrix},$$

which gives more explicitly that

$$A = \begin{pmatrix} \hat{a}_+ \\ \hat{a}_- \end{pmatrix} = \frac{i}{2\tilde{\Omega}} \begin{pmatrix} \sigma_- \hat{z} - \hat{w} \\ -\sigma_+ \hat{z} + \hat{w} \end{pmatrix}.$$

The governing equations can then be expressed as

$$\partial_t A = LA + N(A),$$

with

$$L = \begin{pmatrix} \sigma_+ & 0 \\ 0 & \sigma_- \end{pmatrix},$$
$$N = \frac{i}{2\tilde{\Omega}} \begin{pmatrix} -(\widehat{N_w(z)} + \hat{F}_w) \\ \widehat{N_w(z)} + \hat{F}_w \end{pmatrix}.$$

static _complete_params_with_default(*params*)

This static method is used to complete the *params* container.

compute_freq_diss()

Compute the dissipation frequencies with dissipation only for w.

tendencies_nonlin(*state_fft=None*)

Compute the "nonlinear" tendencies.

Classes

InfoSolverPlate2DDiag(**kargs)	
Simul(params)	Pseudo-spectral solver solving the Föppl-von Kármán equations.

fluidsim.solvers.plate2d.dimensional

Plate2d solver (`fluidsim.solvers.plate2d.dimensional`) Provides:

class fluidsim.solvers.plate2d.dimensional.**Converter**(*C*, *h*, *L=1.0*, *sigma=0.0*)

Bases: object

Converter between dimensional and adimensional variables.

Parameters **C** : number

Defined by the dispersion relation $\omega = Ck^2$.

h : number

Thickness of the plate (in meter).

L : number, optional

Length of the plate (in meter).

sigma : number, optional

Poisson modulus.

Notes

$$C = \frac{Eh^2}{12\rho(1-\sigma^2)}$$

compute_nu4_adim(*nu4*)

Compute the dimensional hyper-viscosity.

compute_nu4_dim(*nu4*)

Compute the adimensional hyper-viscosity.

compute_time_adim(*t*)

Compute the dimensional time.

compute_time_dim(*t*)

Compute the adimensional time.

compute_w_adim(*w*)

Compute the dimensional deformation.

compute_w_dim(*w*)

Compute the adimensional deformation.

compute_z_adim(*z*)

Compute the dimensional deformation.

compute_z_dim(*z*)

Compute the adimensional deformation.

Classes

Converter(C, h[, L, sigma])	Converter between dimensional and adimensional variables.

fluidsim.solvers.ad1d

Advection-diffusion 1D solvers (`fluidsim.solvers.ad1d`)

Provides:

solver	AD1D solver (*fluidsim.solvers.ad1d.solver*)

fluidsim.solvers.ad1d.solver

AD1D solver (`fluidsim.solvers.ad1d.solver`) Provides:

class fluidsim.solvers.ad1d.solver.**Simul**(*params*)

Bases: *fluidsim.base.solvers.base.SimulBase*

Advection-diffusion solver 1D.

Notes

We use a finite difference method with the Crank-Nicolson time scheme to solve the equation

$$\partial_t s + U \partial_x s = D(s),$$

where $d(s)$ is the dissipation term and U is a constant velocity.

static _complete_params_with_default(*params*)

This static method is used to complete the *params* container.

linear_operator()

Compute the linear operator as a matrix.

tendencies_nonlin(*state_phys=None*)

Compute the "nonlinear" tendencies.

Classes

InfoSolverAD1D(**kargs)	
Simul(params)	Advection-diffusion solver 1D.

fluidsim.solvers.waves2d

2D waves solvers (`fluidsim.solvers.waves2d`)

Provides:

solver	2D waves solver (*fluidsim.solvers.waves2d.solver*)

fluidsim.solvers.waves2d.solver

2D waves solver (fluidsim.solvers.waves2d.solver)

class fluidsim.solvers.waves2d.solver.**Simul**(*params*)

Bases: *fluidsim.base.solvers.pseudo_spect.SimulBasePseudoSpectral*

Pseudo-spectral solver for equations of 2D waves.

Notes

This class is dedicated to solve wave 2D equations:

$$\partial_t \hat{f} = \hat{g} - \gamma_f \hat{f},$$
$$\partial_t \hat{g} = -\Omega^2 \hat{f} - \gamma_g \hat{g},$$

This purely linear wave equation can alternatively be written as as $\partial_t X = MX$, with

$$X = \begin{pmatrix} \hat{f} \\ \hat{g} \end{pmatrix} \text{ and } M = \begin{pmatrix} -\gamma_f & 1 \\ -\Omega^2 & -\gamma_g \end{pmatrix},$$

where the three coefficients usually depend on the wavenumber. The eigenvalues are $\sigma_\pm = -\bar{\gamma} \pm i\tilde{\Omega}$, where $\bar{\gamma} = (\gamma_f + \gamma_g)/2$ and

$$\tilde{\Omega} = \Omega \sqrt{1 + \frac{1}{\Omega^2}(\gamma_f \gamma_g - \bar{\gamma}^2)}.$$

The (not normalized) eigenvectors can be expressed as

$$V_\pm = \begin{pmatrix} 1 \\ \sigma_\pm + \gamma_f \end{pmatrix}.$$

The state can be represented by a vector A that verifies $X = VA$, where V is the base matrix

$$V = \begin{pmatrix} 1 & 1 \\ \sigma_+ + \gamma_f & \sigma_- + \gamma_f \end{pmatrix}.$$

The inverse base matrix is given by

$$V^{-1} = \frac{i}{2\tilde{\Omega}} \begin{pmatrix} \sigma_- + \gamma_f & -1 \\ -\sigma_+ - \gamma_f & 1 \end{pmatrix}.$$

static _complete_params_with_default(*params*)

This static method is used to complete the *params* container.

Classes

Simul(params)	Pseudo-spectral solver for equations of 2D waves.

2.4 fluidsim.util

2.4.1 Utilities

CHAPTER 3

Scripts

FluidSim also comes with scripts. They are organised in the following directories:

scripts.launch
scripts.plot_results
scripts.util

3.1 scripts.launch

3.2 scripts.plot_results

3.3 scripts.util

CHAPTER 4

More

4.1 To do list

FluidSim is still in a planning stage so there is still A LOT to do!!

4.1.1 Long term

- Library for 2D pencil decomposition and distributed Fast Fourier Transform: http://www.2decomp.org
- Parallel Three-Dimensional Fast Fourier Transforms: https://code.google.com/p/p3dfft/
- VAPOR is the Visualization and Analysis Platform for Ocean, Atmosphere, and Solar Researchers (https://www.vapor.ucar.edu/)
- VisIt is a free [and open-source], interactive parallel visualization and graphical analysis tool (https://wci.llnl.gov/simulation/computer-codes/visit)

4.1.2 Questions

- String, unicode and byte in Python 2 and 3.

4.1.3 Inline to do items

Todo

Make a nice hierarchy of Operators classes (?)

- Operators
- OperatorsPseudoSpectral1D
- OperatorsPseudoSpectral2D
- OperatorsPseudoSpectral3D
- OperatorsFiniteDiff1D
- OperatorsFiniteDiff2D
- OperatorsFiniteDiff3D
- OperatorsPseudoSpectral1DFiniteDiff1D

- OperatorsPseudoSpectral2DFiniteDiff1D

(The original entry is located in docstring of fluidsim.operators, line 14.)

Todo

- 3D pseudo-spectral operator with parallel fft,
- output and 3D plotting,

(The original entry is located in docstring of fluidsim.solvers.ns3d.solver, line 8.)

Todo

- other solver solving exactly the linear terms.

(The original entry is located in docstring of fluidsim.solvers.plate2d.solver, line 15.)

4.2 Changes

4.2.1 0.0.3a0

Merge with geofluidsim (Ashwin Vishnu Mohanan repository)

- Movies.
- Preprocessing of parameters.
- Less bugs.

4.2.2 0.0.2a1

- Use a cleaner parameter container class (fluiddyn 0.0.8a1).

4.2.3 0.0.2a0

- SetOfVariables inherits from numpy.ndarray.
- The creation of default parameter has been simplified and is done by a class function Simul.create_default_params.

4.2.4 0.0.1a

- Split the package fluiddyn between one base package and specialized packages.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

f

S

Symbols

A

C

D

E

F

G

I

L

N

O

P

Q

R

S

T

U

V