Xcompact3d-toolbox

Release v1.1.0

2021-10-07

Contents

It is a Python package designed to handle the pre and postprocessing of the high-order Navier-Stokes solver Xcompact3d. It aims to help users and code developers to build case-specific solutions with a set of tools and automated processes.

The physical and computational parameters are built on top of traitlets, a framework that lets Python classes have attributes with type checking, dynamically calculated default values, and 'on change' callbacks. In addition to ipywidgets for an user friendly interface.

Data structure is provided by xarray (see Why xarray?), that introduces labels in the form of dimensions, coordinates and attributes on top of raw NumPy-like arrays, which allows for a more intuitive, more concise, and less error-prone developer experience. It integrates tightly with dask for parallel computing and hvplot for interactive data visualization.

Finally, Xcompact3d-toolbox is fully integrated with the new *Sandbox Flow Configuration*. The idea is to easily provide everything that Xcompact3d needs from a Jupyter Notebook, like initial conditions, solid geometry, boundary conditions, and the parameters. It makes life easier for beginners, that can run any new flow configuration without worrying about Fortran and 2decomp. For developers, it works as a rapid prototyping tool, to test concepts and then compare results to validate any future Fortran implementations.

CHAPTER 1

Useful links

- View on GitHub;
- Changelog;
- Suggestions for new features and bug report;
- See what is coming next (Project page).

CHAPTER 2

Installation

It is possible to install using pip:

```
pip install xcompact3d-toolbox
```

There are other dependency sets for extra functionality:

```
pip install xcompact3d-toolbox[visu] # interactive visualization with hvplot and␣
↪others
pip install xcompact3d-toolbox[doc]  # dependencies to build the documentation
pip install xcompact3d-toolbox[dev]  # tools for development
pip install xcompact3d-toolbox[test] # tools for testing
pip install xcompact3d-toolbox[all]  # all the above
```

To install from source, clone de repository:

```
git clone https://github.com/fschuch/xcompact3d_toolbox.git
```

And then install it interactively with pip:

```
cd xcompact3d_toolbox
pip install -e .
```

You can install all dependencies as well:

```
pip install -e .[all]
```

Now, any change you make at the source code will be available at your local installation, with no need to reinstall the package every time.

CHAPTER 3

Examples

- Importing the package:

```
import xcompact3d_toolbox as x3d
```

- Loading the parameters file (both `i3d` and `prm` are supported, see #7) from the disc:

```
prm = x3d.Parameters(loadfile="input.i3d")
prm = x3d.Parameters(loadfile="incompact3d.prm")
```

- Specifying how the binary fields from your simulations are named, for instance:
  - If the simulated fields are named like `ux-000.bin`:

```
prm.dataset.filename_properties.set(
    separator = "-",
    file_extension = ".bin",
    number_of_digits = 3
)
```

  - If the simulated fields are named like `ux0000`:

```
prm.dataset.filename_properties.set(
    separator = "",
    file_extension = "",
    number_of_digits = 4
)
```

- There are many ways to load the arrays produced by your numerical simulation, so you can choose what best suits your post-processing application. All arrays are wrapped into xarray objects, with many useful methods for indexing, comparisons, reshaping and reorganizing, computations and plotting. See the examples:
  - Load one array from the disc:

```
ux = prm.dataset.load_array("ux-0000.bin")
```

  - Load the entire time series for a given variable:

```
ux = prm.dataset["ux"]
```

  - Load all variables from a given snapshot:

```
snapshot = prm.dataset[10]
```

  - Loop through all snapshots, loading them one by one:

```
for ds in prm.dataset:
   # compute something
   vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_derivative("y")
   # write the results to the disc
   prm.dataset.write(data = vort, file_prefix = "w3")
```

  - Or simply load all snapshots at once (if you have enough memory):

```
ds = prm.dataset[:]
```

  - It is possible to produce a new xdmf file, so all data can be visualized on any external tool:

```
prm.dataset.write_xdmf()
```

- User interface for the parameters with IPywidgets:

```
prm = x3d.ParametersGui()
prm
```

CHAPTER 4

Table of Content

4.1 API reference

4.1.1 Parameters

Tools to manipulate the physical and computational parameters. It contains variables and methods designed to be a link between XCompact3d and Python applications for pre and post-processing.

class xcompact3d_toolbox.parameters.**Parameters**(*raise_warning: bool = False*, ***kwargs*)

Bases: *xcompact3d_toolbox.parameters.ParametersBasicParam*, *xcompact3d_toolbox.parameters.ParametersNumOptions*, *xcompact3d_toolbox.parameters.ParametersInOutParam*, *xcompact3d_toolbox.parameters.ParametersScalarParam*, *xcompact3d_toolbox.parameters.ParametersLESModel*, *xcompact3d_toolbox.parameters.ParametersIbmStuff*, *xcompact3d_toolbox.parameters.ParametersALMParam*, *xcompact3d_toolbox.parameters.ParametersExtras*

The physical and computational parameters are built on top of traitlets. It is a framework that lets Python classes have attributes with type checking, dynamically calculated default values, and 'on change' callbacks. In this way, many of the parameters are validated regarding the type, business rules, and the range of values supported by XCompact3d. There are methods to handle the parameters file (`.i3d` and `.prm`).

The parameters are arranged in different classes, but just for organization purposes, this class inherits from all of them.

In addition, there are ipywidgets for a friendly user interface, see *xcompact3d_toolbox.gui.ParametersGui*.

After that, it is time to read the binary arrays produced by XCompact3d and also to write a new xdmf file, so the binary fields can be opened in any external visualization tool. See more details in *xcompact3d_toolbox.parameters.ParametersExtras.dataset*.

Note: This is a work in progress, not all parameters are covered yet.

__init__(*raise_warning: bool = False*, ***kwargs*)

Initializes the Parameters Class.

Parameters

- **raise_warning** (*bool, optional*) – Raise a warning instead of an error if an invalid parameter is found. By default False.
- ****kwargs** – Keyword arguments for valid attributes, like `nx`, `re` and so on.

Raises `KeyError` – Exception is raised when an Keyword arguments is not a valid attribute.

Examples

There are a few ways to initialize the class.

First, calling it with no arguments initializes all variables with default value:

```
>>> prm = xcompact3d_toolbox.Parameters()
```

It is possible to set any value afterwards:

```
>>> prm.re = 1e6
>>> prm.set(
...     iibm = 0,
...     p_row = 4,
...     p_col = 2,
... )
```

Second, we can specify some values, and let the missing ones be initialized with default value:

```
>>> prm = x3d.Parameters(
...     filename = 'example.i3d',
...     itype = 12,
...     nx = 257,
...     ny = 129,
...     nz = 32,
...     xlx = 15.0,
...     yly = 10.0,
...     zlz = 3.0,
...     nclx1 = 2,
...     nclxn = 2,
...     ncly1 = 1,
...     nclyn = 1,
...     nclz1 = 0,
...     nclzn = 0,
...     re = 300.0,
...     init_noise = 0.0125,
...     dt = 0.0025,
...     ilast = 45000,
...     ioutput = 200,
...     iprocessing = 50
... )
```

And finally, it is possible to read the parameters from the disc:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile = 'example.i3d')
```

It also supports the previous parameters file format (see #7):

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile = 'incompact3d.prm')
```

__repr__()
Return repr(self).

__str__()
Representation of the parameters class, similar to the representation of the `.i3d` file.

from_file(*filename: str = None*, *raise_warning: bool = False*) → None
Loads the attributes from the parameters file.

It also includes support for the previous format `prm` (see #7).

Parameters

- **filename** (*str, optional*) – The filename for the parameters file. If None, it uses the filename specified in the class (default is `None`).
- **raise_warning** (*bool, optional*) – Raise a warning instead of an error if an invalid parameter is found. By default False.

Raises

- `IOError` – If the file format is not `i3d` or `prm`.
- `KeyError` – Exception is raised when an attributes is invalid.

Examples

```
>>> prm = xcompact3d_toolbox.Parameters(filename = 'example.i3d')
>>> prm.load()
```

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.load('example.i3d')
```

or just:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile = 'example.i3d')
>>> prm = xcompact3d_toolbox.Parameters(loadfile = 'incompact3d.prm')
```

from_string(*string: str*, *raise_warning: bool = False*) → None
Loads the attributes from a string.

Parameters

- **filename** (*str, optional*) – The filename for the parameters file. If None, it uses the filename specified in the class (default is `None`).
- **raise_warning** (*bool, optional*) – Raise a warning instead of an error if an invalid parameter is found. By default False.

Raises `KeyError` – Exception is raised when an attributes is invalid.

get_boundary_condition(*variable_name: str*) → dict
This method returns the appropriate boundary parameters that are expected by the derivatives methods.

Parameters **variable_name** (*str*) – Variable name. The supported options are ux, uy, uz, pp and phi, otherwise the method returns the default option.

Returns A dict containing the boundary conditions for the variable specified.

Return type dict of dict

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.get_boundary_condition('ux')
{'x': {'ncl1': 1, 'ncln': 1, 'npaire': 0},
'y': {'ncl1': 1, 'ncln': 2, 'npaire': 1, 'istret': 0, 'beta': 0.75},
'z': {'ncl1': 0, 'ncln': 0, 'npaire': 1}}
```

It is possible to store this information as an attribute in any xarray.DataArray:

```
>>> DataArray.attrs['BC'] = prm.get_boundary_condition('ux')
```

So the correct boundary conditions will be used to compute the derivatives:

```
>>> DataArray.x3d.first_derivative('x')
>>> DataArray.x3d.second_derivative('x')
```

get_mesh(*refined_for_ibm: bool = False*) → dict

Get mesh the three-dimensional coordinate system. The coordinates are stored in a dictionary. It supports mesh refinement in **y** when *ParametersBasicParam.istret* $\neq 0$.

Parameters **refined_for_ibm** (*bool*) – If True, it returns a refined mesh as a function of *ParametersIbmStuff.nraf* (default is False).

Returns It contains the mesh point locations at three dictionary keys, for **x**, **y** and **z**.

Return type dict of numpy.ndarray

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.get_mesh()
{'x': array([0.    , 0.0625, 0.125 , 0.1875, 0.25  , 0.3125, 0.375 , 0.4375,
       0.5   , 0.5625, 0.625 , 0.6875, 0.75  , 0.8125, 0.875 , 0.9375,
       1.    ]),
 'y': array([0.    , 0.0625, 0.125 , 0.1875, 0.25  , 0.3125, 0.375 , 0.4375,
       0.5   , 0.5625, 0.625 , 0.6875, 0.75  , 0.8125, 0.875 , 0.9375,
       1.    ]),
 'z': array([0.    , 0.0625, 0.125 , 0.1875, 0.25  , 0.3125, 0.375 , 0.4375,
       0.5   , 0.5625, 0.625 , 0.6875, 0.75  , 0.8125, 0.875 , 0.9375,
       1.    ])}
```

load(**arg*, ***kwarg*) → None

An alias for *Parameters.from_file*

set(*raise_warning: bool = False*, ***kwargs*) → None

Set a new value for any parameter after the initialization.

Parameters

- **raise_warning** (*bool, optional*) – Raise a warning instead of an error if an invalid parameter is found. By default False.
- ****kwargs** – Keyword arguments for valid attributes, like `nx`, `re` and so on.

Raises `KeyError` – Exception is raised when an Keyword arguments is not a valid attribute.

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.set(
...     iibm = 0,
...     p_row = 4,
...     p_col = 2,
... )
```

write (*filename: str = None*) → None

Write all valid attributes to an `i3d` file.

An attribute is considered valid if it has a `tag` named `group`, witch assigns it to the respective namespace at the `i3d` file.

Parameters **filename** (*str, optional*) – The filename for the `i3d` file. If None, it uses the filename specified in the class (default is `None`).

Examples

```
>>> prm = xcompact3d_toolbox.Parameters(
...     filename = 'example.i3d',
...     nx = 101,
...     ny = 65,
...     nz = 11,
...     # and so on...
... )
>>> prm.write()
```

or just:

```
>>> prm.write('example.i3d')
```

class xcompact3d_toolbox.parameters.**ParametersALMParam**

Bases: `traitlets.traitlets.HasTraits`

__init__()

Initialize self. See help(type(self)) for accurate signature.

class xcompact3d_toolbox.parameters.**ParametersBasicParam**

Bases: `traitlets.traitlets.HasTraits`

__init__()

Initialize self. See help(type(self)) for accurate signature.

beta

Refinement factor in **y**.

Notes

Only necessary if *istret* $\neq 0$.

Type float

dt
Time step (Δt).

Type float

gravx
Component of the unitary vector pointing in the gravity's direction.

Type float

gravy
Component of the unitary vector pointing in the gravity's direction.

Type float

gravz
Component of the unitary vector pointing in the gravity's direction.

Type float

ifirst
The number for the first iteration.

Type int

iibm

- 0 - Off (default);
- 1 - On with direct forcing method, i.e., it sets velocity to zero inside the solid body;
- 2 - On with alternating forcing method, i.e, it uses Lagrangian Interpolators to define the velocity inside the body and imposes no-slip condition at the solid/fluid interface;
- 3 - Cubic Spline Reconstruction;

Any option greater than zero activates the namespace **ibmstuff**, for variables like *ParametersIbmStuff.nobjmax* and *ParametersIbmStuff.nraf*.

Type int

Type Enables Immersed Boundary Method (IBM)

iin

- 0 - No random noise (default);
- 1 - Random noise with amplitude of *init_noise*;
- 2 - Random noise with fixed seed (important for reproducibility, development and debugging) and amplitude of *init_noise*;
- 3 - Read inflow planes.

Notes

The exactly behavior may be different according to each flow configuration.

Type int

Type Defines perturbation at the initial condition

ilast
The number for the last iteration.

Type int

ilesmod

- 0 - No (also forces *ParametersNumOptions.nu0nu* and *ParametersNumOptions.cnu* to 4.0 and 0.44, respectively);
- 1 - Yes (also activates the namespace **LESModel** (see *ParametersLESModel*).

Type int

Type Enables Large-Eddy methodologies

inflow_noise
Random number amplitude at inflow boundary (where $x = 0$).

Notes

Only necessary if *nclx1* is equal to 2.

Type float

init_noise
Random number amplitude at initial condition.

Notes

Only necessary if *iin* $\neq 0$. The exactly behavior may be different according to each flow configuration.

Type float

ipost

- 0 - No;
- 1 - Yes (default).

Note: The computation for each case is specified at the `BC.<flow-configuration>.f90` file.

Type int

Type Enables online postprocessing at a frequency *ParametersInOutParam.iprocessing*

istret

- 0 - No refinement (default);
- 1 - Refinement at the center;
- 2 - Both sides;
- 3 - Just near the bottom.

Notes

See *beta*.

Type int

Type Controls mesh refinement in **y**

itype

Sets the flow configuration, each one is specified in a different `BC.<flow-configuration>.f90` file (see Xcompact3d/src), they are:

- 0 - User configuration;
- 1 - Turbidity Current in Lock-Release;
- 2 - Taylor-Green Vortex;
- 3 - Periodic Turbulent Channel;
- 4 - Periodic Hill
- 5 - Flow around a Cylinder;
- 6 - Debug Schemes (for developers);
- 7 - Mixing Layer;
- 8 - Jet;
- 9 - Turbulent Boundary Layer;
- 10 - ABL;
- 11 - Uniform;
- 12 - *Sandbox*.

Type int

ivisu

- 0 - No;
- 1 - Yes (default).

Type int

Type Enables store snapshots at a frequency *ParametersInOutParam.ioutput*

nclx1

- 0 - Periodic;
- 1 - Free-slip;
- 2 - Inflow.

Type int

Type Boundary condition for velocity field where $x = 0$, the options are

nclxn

- 0 - Periodic;

- 1 - Free-slip;
- 2 - Convective outflow.

Type int

Type Boundary condition for velocity field where $x = L_x$, the options are

ncly1

- 0 - Periodic;
- 1 - Free-slip;
- 2 - No-slip.

Type int

Type Boundary condition for velocity field where $y = 0$, the options are

nclyn

- 0 - Periodic;
- 1 - Free-slip;
- 2 - No-slip.

Type int

Type Boundary condition for velocity field where $y = L_y$, the options are

nclz1

- 0 - Periodic;
- 1 - Free-slip;
- 2 - No-slip.

Type int

Type Boundary condition for velocity field where $z = 0$, the options are

nclzn

- 0 - Periodic;
- 1 - Free-slip;
- 2 - No-slip.

Type int

Type Boundary condition for velocity field where $z = L_z$, the options are

numscalar
Number of scalar fraction, which can have different properties. Any option greater than zero activates the namespace *ParametersScalarParam*.

Type int

nx
Number of mesh points.

Notes

See *xcompact3d_toolbox.mesh.Coordinate.possible_grid_size* for recommended grid sizes.

Type int

ny
Number of mesh points.

Notes

See *xcompact3d_toolbox.mesh.Coordinate.possible_grid_size* for recommended grid sizes.

Type int

nz
Number of mesh points.

Notes

See *xcompact3d_toolbox.mesh.Coordinate.possible_grid_size* for recommended grid sizes.

Type int

p_col
Defines the domain decomposition for (large-scale) parallel computation.

Notes

The product `p_row * p_col` must be equal to the number of computational cores where XCompact3d will run. More information can be found at 2DECOMP&FFT.

`p_row = p_col = 0` activates auto-tunning.

Type int

p_row
Defines the domain decomposition for (large-scale) parallel computation.

Notes

The product `p_row * p_col` must be equal to the number of computational cores where XCompact3d will run. More information can be found at 2DECOMP&FFT.

`p_row = p_col = 0` activates auto-tunning.

Type int

re
Reynolds number (Re).

Type float

xlx
Domain size.

Type float

yly
Domain size.

Type float

zlz
Domain size.

Type float

class xcompact3d_toolbox.parameters.**ParametersExtras**
Bases: traitlets.traitlets.HasTraits

Extra utilities that are not present at the parameters file, but are usefull for Python applications.

__init__()
Initialize self. See help(type(self)) for accurate signature.

dataset
An object that reads and writes the raw binary files from XCompact3d on-demand.

Notes

All arrays are wrapped into Xarray objects (xarray.DataArray or xarray.Dataset), take a look at xarray's documentation, specially, see Why xarray? Xarray has many useful methods for indexing, comparisons, reshaping and reorganizing, computations and plotting.

Consider using hvPlot to explore your data interactively, see how to plot Gridded Data.

Examples

The first step is specify the filename properties. If the simulated fields are named like ux-000.bin, they are in the default configuration, there is no need to specify filename properties. But just in case, it would be like:

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.dataset.filename_properties.set(
...     separator = "-",
...     file_extension = ".bin",
...     number_of_digits = 3
... )
```

If the simulated fields are named like ux0000, the parameters are:

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.dataset.filename_properties.set(
...     separator = "",
...     file_extension = "",
...     number_of_digits = 4
... )
```

Data type is defined by xcompact3d_toolbox.param:

```
>>> import numpy
>>> xcompact3d_toolbox.param["mytype] = numpy.float64 # if double precision
>>> xcompact3d_toolbox.param["mytype] = numpy.float32 # if single precision
```

Now it is possible to customize the way the dataset will be handled:

```
>>> prm.dataset.set(
...     data_path = "./data/",
...     drop_coords = "",
...     set_of_variables = {"ux", "uy", "uz"},
...     snapshot_step = "ioutput",
...     snapshot_counting = "ilast",
...     stack_scalar = True,
...     stack_velocity = False,
... )
```

Note: For convenience, `data_path` is set as `"./data/"` relative to the `filename` of the parameters file when creating a new instance of *Parameters* (i.e., if `filename = "./example/input.i3d"` then `data_path = "./example/data/"`).

There are many ways to load the arrays produced by your numerical simulation, so you can choose what best suits your post-processing application. See the examples:

- Load one array from the disc:

```
>>> ux = prm.dataset.load_array("ux-0000.bin")
```

- Load the entire time series for a given variable:

```
>>> ux = prm.dataset.load_time_series("ux")
>>> uy = prm.dataset.load_time_series("uy")
>>> uz = prm.dataset.load_time_series("uz")
```

or just:

```
>>> ux = prm.dataset["ux"]
>>> uy = prm.dataset["uy"]
>>> uz = prm.dataset["uz"]
```

You can organize them using a dataset:

```
>>> dataset = xarray.Dataset()
>>> for var in "ux uy uz".split():
...     dataset[var] = prm.dataset[var]
```

- Load all variables from a given snapshot:

```
>>> snapshot = prm.dataset.load_snapshot(10)
```

or just:

```
>>> snapshot = prm.dataset[10]
```

- Loop through all snapshots, loading them one by one:

```
>>> for ds in prm.dataset:
...     vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_
↪derivative("y")
...     prm.dataset.write(data = vort, file_prefix = "w3")
```

- Loop through some snapshots, loading them one by one, with the same arguments of a classic Python `range`, for instance, from 0 to 100 with a step of 5:

```
>>> for ds in prm.dataset(0, 101, 5):
...     vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_
↪derivative("y")
...     prm.dataset.write(data = vort, file_prefix = "w3")
```

- Or simply load all snapshots at once (if you have enough memory):

```
>>> ds = prm.dataset[:]
```

And finally, it is possible to produce a new xdmf file, so all data can be visualized on any external tool:

```
>>> prm.dataset.write_xdmf()
```

Type *xcompact3d_toolbox.io.Dataset*

dx
Mesh resolution.

Type float

dy
Mesh resolution.

Type float

dz
Mesh resolution.

Type float

filename
Filename for the parameters file.

Type str

mesh
Mesh object.

Type *xcompact3d_toolbox.mesh.Mesh3D*

ncores
Number of computational cores where XCompact3d will run.

Type int

size
Auxiliar variable indicating the demand for storage.

Type str

class xcompact3d_toolbox.parameters.**ParametersIbmStuff**
Bases: traitlets.traitlets.HasTraits

__init__()
Initialize self. See help(type(self)) for accurate signature.

izap
How many points to skip for reconstruction ranging from 0 to 3, the recommended is 1.

Type int

nobjmax
Maximum number of objects in any direction. It is defined automatically at `gene_epsi_3D`.

Type int

npif
Number of Points for the reconstruction ranging from 1 to 3, the recommended is 2.

Type int

nraf
Level of refinement to find the surface of the immersed object, when *ParametersBasicParam.iibm* is equal to 2.

Type int

class xcompact3d_toolbox.parameters.**ParametersInOutParam**
Bases: traitlets.traitlets.HasTraits

__init__()
Initialize self. See help(type(self)) for accurate signature.

icheckpoint
Frequency for writing restart file.

Type int

ioutput
Frequency for visualization (3D snapshots).

Type int

iprocessing
Frequency for online postprocessing.

Type int

irestart
Reads initial flow field if equals to 1.

Type int

nvisu
Size for visual collection.

Type int

class xcompact3d_toolbox.parameters.**ParametersLESModel**
Bases: traitlets.traitlets.HasTraits

__init__()
Initialize self. See help(type(self)) for accurate signature.

iwall
type: int

jles

- 0 - No model (DNS);
- 1 - Phys Smag;
- 2 - Phys WALE;
- 3 - Phys dyn. Smag;
- 4 - iSVV.

Type int

Type Chooses LES model, they are

maxdsmagcst
type: float

nSmag
type: float

smagcst
type: float

smagwalldamp
type: int

walecst
type: float

class xcompact3d_toolbox.parameters.**ParametersNumOptions**
Bases: traitlets.traitlets.HasTraits

__init__()
Initialize self. See help(type(self)) for accurate signature.

cnu
Ratio between hyperviscosity at $k_m = 2/3\pi$ and $k_c = \pi$.

Type float

ifirstder

- 1 - 2nd central;
- 2 - 4th central;
- 3 - 4th compact;
- 4 - 6th compact (default).

Type int

Type Scheme for first order derivative

iimplicit

- 0 - Off (default);
- 1 - With backward Euler for Y diffusion;
- 2 - With CN for Y diffusion.

Type int

Type Time integration scheme

isecondder

- 1 - 2nd central;
- 2 - 4th central;
- 3 - 4th compact;
- 4 - 6th compact (default);

- 5 - Hyperviscous 6th.

Type int

Type Scheme for second order derivative

itimescheme

- 1 - Forwards Euler;
- 2 - Adams-bashforth 2;
- 3 - Adams-bashforth 3 (default);
- 4 - Adams-bashforth 4 (Not Implemented);
- 5 - Runge-kutta 3;
- 6 - Runge-kutta 4 (Not Implemented).

Type int

Type Time integration scheme

nu0nu
Ratio between hyperviscosity/viscosity at nu.

Type float

class xcompact3d_toolbox.parameters.**ParametersScalarParam**
Bases: traitlets.traitlets.HasTraits

__init__()
Initialize self. See help(type(self)) for accurate signature.

cp
Initial concentration(s).

Type list of float

nclxS1

- 0 - Periodic;
- 1 - No-flux;
- 2 - Inflow.

Type int

Type Boundary condition for scalar field(s) where $x = 0$, the options are

nclxSn

- 0 - Periodic;
- 1 - No-flux;
- 2 - Convective outflow.

Type int

Type Boundary condition for scalar field(s) where $x = L_x$, the options are

nclyS1

- 0 - Periodic;
- 1 - No-flux;
- 2 - Dirichlet.

Type int

Type Boundary condition for scalar field(s) where $y = 0$, the options are

nclySn

- 0 - Periodic;
- 1 - No-flux;
- 2 - Dirichlet.

Type int

Type Boundary condition for scalar field(s) where $y = L_y$, the options are

nclzS1

- 0 - Periodic;
- 1 - No-flux;
- 2 - Dirichlet.

Type int

Type Boundary condition for scalar field(s) where $z = 0$, the options are

nclzSn

- 0 - Periodic;
- 1 - No-flux;
- 2 - Dirichlet.

Type int

Type Boundary condition for scalar field(s) where $z = L_z$, the options are

ri
Richardson number(s).

Type `list` of `float`

sc
Schmidt number(s).

Type `list` of `float`

scalar_lbound
Lower scalar bound(s), for clipping methodology.

Type `list` of `float`

scalar_ubound
Upper scalar bound(s), for clipping methodology.

Type `list` of `float`

uset
Settling velocity(s).

Type list of float

4.1.2 Parameters - GUI

Manipulate the physical and computational parameters, just like *xcompact3d_toolbox.parameters.Parameters*, but with ipywidgets.

class xcompact3d_toolbox.gui.**ParametersGui**(**kwargs*)
Bases: *xcompact3d_toolbox.parameters.Parameters*

This class is derivated from *xcompact3d_toolbox.parameters.Parameters*, including all its features. In addition, there is a two way link between the parameters and their widgets. Control them with code and/or with the graphical user interface.

__call__(**args) -> Type(widgets.VBox)*
Returns widgets on demand.

Parameters ***args** (*str*) – Name(s) for the desired widget(s).

Returns Widgets for an user friendly interface.

Return type ipywidgets.VBox

Examples

```
>>> prm = xcompact3d_toolbox.ParametersGui()
>>> prm('nx', 'xlx', 'dx', 'nclx1', 'nclxn')
```

__init__(**kwargs*)
Initializes the Parameters Class.

Parameters ****kwargs** – Keyword arguments for *xcompact3d_toolbox.parameters.Parameters*.

link_widgets() → None
Creates a two-way link between the value of an attribute and its widget. This method is called at initialization, but provides an easy way to link any new variable.

Examples

```
>>> prm = xcompact3d_toolbox.ParametersGui(loadfile = 'example.i3d')
>>> prm.link_widgets()
```

4.1.3 Mesh

Objects to handle the coordinates and coordinate system. Note they are an atribute at *xcompact3d_toolbox.parameters.ParametersExtras*, so they work together with all the other parameters. They are presented here for reference.

class xcompact3d_toolbox.mesh.**Coordinate**(**kwargs*)
Bases: traitlets.traitlets.HasTraits

A coordinate.

Thanks to traitlets, the attributes can be type checked, validated and also trigger 'on change' callbacks. It means that:

- `grid_size` is validated to just accept the values expected by XCompact3d (see *xcompact3d_toolbox.mesh.Coordinate.possible_grid_size*);
- `delta` is updated after any change on `grid_size` or `length`;
- `length` is updated after any change on `delta` (`grid_size` remais constant);
- `grid_size` is reduced automatically by 1 when `is_periodic` changes to `True` and it is added by 1 when `is_periodic` changes back to `False` (see *xcompact3d_toolbox.mesh.Coordinate.possible_grid_size*);

All these functionalities aim to make a user-friendly interface, where the consistency between different coordinate parameters is ensured even when they change at runtime.

Parameters

- **length** (*float*) – Length of the coordinate (default is 1.0).
- **grid_size** (*int*) – Number of mesh points (default is 17).
- **delta** (*float*) – Mesh resolution (default is 0.0625).
- **is_periodic** (*bool*) – Specifies if the boundary condition is periodic (True) or not (False) (default is False).

Notes

There is no need to specify both `length` and `delta`, because they are a function of each other, the missing value is automatically computed from the other.

Returns Coordinate

Return type *xcompact3d_toolbox.mesh.Coordinate*

__array__() -> *Type(np.ndarray)*

This method makes the coordinate automatically work as a numpy like array in any function from numpy.

Returns A numpy array.

Return type `numpy.ndarray`

Examples

```
>>> from xcompact3d_toolbox.mesh import Coordinate
>>> import numpy
>>> coord = Coordinate(length = 1.0, grid_size = 9)
>>> numpy.sin(coord)
array([0.        , 0.12467473, 0.24740396, 0.36627253, 0.47942554,
       0.58509727, 0.68163876, 0.7675435 , 0.84147098])
>>> numpy.cos(coord)
array([1.        , 0.99219767, 0.96891242, 0.93050762, 0.87758256,
        0.81096312, 0.73168887, 0.64099686, 0.54030231])
```

__init__(**kwargs)

Initializes the Coordinate class.

Parameters ****kwargs** – Keyword arguments for attributes, like `grid_size`, `length` and so on.

Raises KeyError – Exception is raised when an Keyword arguments is not a valid attribute.

Examples

```
>>> from xcompact3d_toolbox.mesh import Coordinate
>>> coord = Coordinate(length = 1.0, grid_size = 9, is_periodic = False)
```

__len__()

Make the coordinate work with the Python function len.

Returns Coordinate size (grid_size)

Return type int

Examples

```
>>> from xcompact3d_toolbox.mesh import Coordinate
>>> coord = Coordinate(grid_size = 9)
>>> len(coord)
9
```

__repr__()

Return repr(self).

possible_grid_size

Possible values for grid size.

Due to restrictions at the FFT library, they must be equal to:

$$n = 2^{1+a} \times 3^b \times 5^c,$$

if the coordinate is periodic, and:

$$n = 2^{1+a} \times 3^b \times 5^c + 1,$$

otherwise, where a, b and c are non negative integers.

Aditionally, the derivative's stencil imposes that $n \geq 8$ if periodic and $n \geq 9$ otherwise.

Returns Possible values for grid size

Return type list

Notes

There is no upper limit, as long as the restrictions are satisfied.

Examples

```
>>> from xcompact3d_toolbox.mesh import Coordinate
>>> coordinate(is_periodic = True).possible_grid_size
[8, 10, 12, 16, 18, 20, 24, 30, 32, 36, 40, 48, 50, 54, 60, 64, 72, 80,
 90, 96, 100, 108, 120, 128, 144, 150, 160, 162, 180, 192, 200, 216, 240,
 250, 256, 270, 288, 300, 320, 324, 360, 384, 400, 432, 450, 480, 486,
```

(continues on next page)

(continued from previous page)

```
 500, 512, 540, 576, 600, 640, 648, 720, 750, 768, 800, 810, 864, 900,
 960, 972, 1000, 1024, 1080, 1152, 1200, 1250, 1280, 1296, 1350, 1440,
 1458, 1500, 1536, 1600, 1620, 1728, 1800, 1920, 1944, 2000, 2048, 2160,
 2250, 2304, 2400, 2430, 2500, 2560, 2592, 2700, 2880, 2916, 3000, 3072,
 3200, 3240, 3456, 3600, 3750, 3840, 3888, 4000, 4050, 4096, 4320, 4374,
 4500, 4608, 4800, 4860, 5000, 5120, 5184, 5400, 5760, 5832, 6000, 6144,
 6250, 6400, 6480, 6750, 6912, 7200, 7290, 7500, 7680, 7776, 8000, 8100,
 8192, 8640, 8748, 9000]
>>> coordinate(is_periodic = False).possible_grid_size
[9, 11, 13, 17, 19, 21, 25, 31, 33, 37, 41, 49, 51, 55, 61, 65, 73, 81,
 91, 97, 101, 109, 121, 129, 145, 151, 161, 163, 181, 193, 201, 217, 241,
 251, 257, 271, 289, 301, 321, 325, 361, 385, 401, 433, 451, 481, 487,
 501, 513, 541, 577, 601, 641, 649, 721, 751, 769, 801, 811, 865, 901,
 961, 973, 1001, 1025, 1081, 1153, 1201, 1251, 1281, 1297, 1351, 1441,
 1459, 1501, 1537, 1601, 1621, 1729, 1801, 1921, 1945, 2001, 2049, 2161,
 2251, 2305, 2401, 2431, 2501, 2561, 2593, 2701, 2881, 2917, 3001, 3073,
 3201, 3241, 3457, 3601, 3751, 3841, 3889, 4001, 4051, 4097, 4321, 4375,
 4501, 4609, 4801, 4861, 5001, 5121, 5185, 5401, 5761, 5833, 6001, 6145,
 6251, 6401, 6481, 6751, 6913, 7201, 7291, 7501, 7681, 7777, 8001, 8101,
 8193, 8641, 8749, 9001]
```

set (***kwargs*) → None

Set a new value for any parameter after the initialization.

Parameters ****kwargs** – Keyword arguments for attributes, like `grid_size`, `length` and so on.

Raises `KeyError` – Exception is raised when an Keyword arguments is not a valid attribute.

Examples

```
>>> from xcompact3d_toolbox.mesh import Coordinate
>>> coord = Coordinate()
>>> coord.set(length = 1.0, grid_size = 9, is_periodic = False)
```

size

An alias for `grid_size`.

Returns Grid size

Return type int

vector

Construct a vector with `numpy.linspace` and return it.

Returns Numpy array

Return type `numpy.ndarray`

class xcompact3d_toolbox.mesh.**Mesh3D**(***kwargs*)

Bases: `traitlets.traitlets.HasTraits`

A three-dimensional coordinate system

Parameters

- **x** (*xcompact3d_toolbox.mesh.Coordinate*) – Streamwise coordinate
- **y** (*xcompact3d_toolbox.mesh.StretchedCoordinate*) – Vertical coordinate

- **z** (*xcompact3d_toolbox.mesh.Coordinate*) – Spanwise coordinate

Notes

`mesh` is in fact an atribute of *xcompact3d_toolbox.parameters.ParametersExtras*, so there is no need to initialize it manually for most of the common use cases. The features of each coordinate are copled by a two-way link with their corresponding values at the Parameters class. For instance, the length of each of them is copled to `xlx`, `yly` and `zlz`, grid size to `nx`, `ny` and `nz` and so on.

Returns Coordinate system

Return type *xcompact3d_toolbox.mesh.Mesh3D*

__init__(***kwargs*)
Initializes the 3DMesh class.

Parameters ****kwargs** – Keyword arguments for each coordinate (x, y and z), containing a `dict` with the parameters for them, like `grid_size`, `length` and so on.

Raises `KeyError` – Exception is raised when an Keyword arguments is not a valid coordinate.

Examples

```
>>> from xcompact3d_toolbox.mesh import Mesh3D
>>> mesh = Mesh3D(
...     x = dict(length = 4.0, grid_size = 65, is_periodic = False),
...     y = dict(length = 1.0, grid_size = 17, is_periodic = False, istret =␣
↪0),
...     z = dict(length = 1.0, grid_size = 16, is_periodic = True)
... )
```

__len__()
Make the coordinate work with the Python function `len`.

Returns Mesh size is calculated by multiplying the size of the three coordinates

Return type int

__repr__()
Return repr(self).

copy()
Return a copy of the Mesh3D object.

drop(**args*) → dict
Get the coordinates in a dictionary, where the keys are their names and the values are their vectors. It is possible to drop any of the coordinates in case they are needed to process planes. For instance:

- Drop `x` if working with `yz` planes;
- Drop `y` if working with `xz` planes;
- Drop `z` if working with `xy` planes.

Parameters ***args** (*str or list of str*) – Name of the coordenate(s) to be dropped

Raises `KeyError` – Exception is raised when an Keyword arguments is not a valid attribute.

Returns A dict containing the desired coordinates

Return type `dict` of `numpy.ndarray`

get() → dict
Get the three coordinates in a dictionary, where the keys are their names (x, y and z) and the values are their vectors.

Raises KeyError – Exception is raised when an Keyword arguments is not a valid attribute.

Returns A dict containing the coordinates

Return type dict of numpy.ndarray

Notes

It is an alias for `Mesh3D.drop(None)`.

set(***kwargs*) → None
Set new values for any of the coordinates after the initialization.

Parameters ****kwargs** – Keyword arguments for each coordinate (x, y and z), containing a dict with the parameters for them, like grid_size, length and so on.

Raises KeyError – Exception is raised when an Keyword arguments is not a valid attribute.

Examples

```
>>> from xcompact3d_toolbox.mesh import Mesh3D
>>> mesh = Mesh3D()
>>> mesh.set(
...     x = dict(length = 4.0, grid_size = 65, is_periodic = False),
...     y = dict(length = 1.0, grid_size = 17, is_periodic = False, istret =␣
↪0),
...     z = dict(length = 1.0, grid_size = 16, is_periodic = True)
... )
```

size
Mesh size

Returns Mesh size is calculated by multiplying the size of the three coordinates

Return type int

class xcompact3d_toolbox.mesh.**StretchedCoordinate**(***kwargs*)
Bases: *xcompact3d_toolbox.mesh.Coordinate*

Another coordinate, as a subclass of *Coordinate*. It includes parameters and methods to handle stretched coordinates, which is employed by XCompact3d at the vertical dimension y.

Parameters

- **length** (*float*) – Length of the coordinate (default is 1.0).
- **grid_size** (*int*) – Number of mesh points (default is 17).
- **delta** (*float*) – Mesh resolution (default is 0.0625).
- **is_periodic** (*bool*) – Specifies if the boundary condition is periodic (True) or not (False) (default is False).
- **istret** (*int*) – Type of mesh refinement:
  - 0 - No refinement (default);
  - 1 - Refinement at the center;

  - 2 - Both sides;
  - 3 - Just near the bottom.
- **beta** (*float*) – Refinement parameter.

Notes

There is no need to specify both `length` and `delta`, because they are a function of each other, the missing value is automatically computed from the other.

Returns Stretched coordinate

Return type *xcompact3d_toolbox.mesh.StretchedCoordinate*

__array__()
This method makes the coordinate automatically work as a numpy like array in any function from numpy.

Returns A numpy array.

Return type numpy.ndarray

Examples

```
>>> from xcompact3d_toolbox.mesh import StretchedCoordinate
>>> import numpy
>>> coord = StretchedCoordinate(length = 1.0, grid_size = 9)
>>> numpy.sin(coord)
array([0.        , 0.12467473, 0.24740396, 0.36627253, 0.47942554,
       0.58509727, 0.68163876, 0.7675435 , 0.84147098])
>>> numpy.cos(coord)
array([1.        , 0.99219767, 0.96891242, 0.93050762, 0.87758256,
        0.81096312, 0.73168887, 0.64099686, 0.54030231])
```

__repr__()
Return repr(self).

4.1.4 Reading and writing files

Usefull objects to read and write the binary fields produced by XCompact3d.

class xcompact3d_toolbox.io.**Dataset**(***kwargs*)
Bases: traitlets.traitlets.HasTraits

An object that reads and writes the raw binary files from XCompact3d on-demand.

Parameters

- **data_path** (*str*) – The path to the folder where the binary fields are located (default is `"./data/"`). .. note :: the default `"./data/"` is relative to the path to the parameters

  file when initialized from *xcompact3d_toolbox.parameters.ParametersExtras*.
- **drop_coords** (*str*) – If working with two-dimensional planes, specify which of the coordinates should be dropped, i.e., `"x"`, `"y"` or `"z"`, or leave it empty for 3D fields (default is `""`).

- **filename_properties** (*FilenameProperties*) – Specifies filename properties for the binary files, like the separator, file extension and number of digits.
- **set_of_variables** (*set*) – The methods in this class will try to find all variables per snapshot, use this parameter to work with just a few specified variables if you need to speedup your application (default is an empty set).
- **snapshot_counting** (*str*) – The parameter that controls the number of timesteps used to produce the datasets (default is `"ilast"`).
- **snapshot_step** (*str*) – The parameter that controls the number of timesteps between each snapshot, it is often `"ioutput"` or `"iprocessing"` (default is `"ioutput"`).
- **stack_scalar** (*bool*) – When `True`, the scalar fields will be stacked in a new coordinate `n`, otherwise returns one array per scalar fraction (default is `False`).
- **stack_velocity** (*bool*) – When `True`, the velocity will be stacked in a new coordinate `i` , otherwise returns one array per velocity component (default is `False`).

Notes

- *Dataset* is in fact an atribute of *xcompact3d_toolbox.parameters.ParametersExtras*, so there is no need to initialize it manually for most of the common use cases.
- All arrays are wrapped into Xarray objects (`xarray.DataArray` or `xarray.Dataset`), take a look at xarray's documentation, specially, see Why xarray? Xarray has many useful methods for indexing, comparisons, reshaping and reorganizing, computations and plotting.
- Consider using hvPlot to explore your data interactively, see how to plot Gridded Data.

__call__(**args*) → Type[xarray.core.dataset.Dataset]

Yields selected snapshots, so the application can iterate over them, loading one by one, with the same arguments of a classic Python `range`.

Parameters ***args** (*int*) – Same arguments used for a `range`.

Yields `xarray.Dataset` – Dataset containing the arrays loaded from the disc with the appropriate dimensions, coordinates and attributes.

Examples

Initial setup:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="input.i3d")
>>> prm.dataset.set(
...     filename_properties=dict(
...         separator="-",
...         file_extension=".bin",
...         number_of_digits=3
...     ),
...     stack_scalar=True,
...     stack_velocity=True,
... )
```

Iterate over some snapshots, loading them one by one, with the same arguments of a classic Python `range`, for instance, from 0 to 100 with a step of 5:

```
>>> for ds in prm.dataset(0, 101, 5):
...     vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_derivative("y
↪")
...     prm.dataset.write(data = vort, file_prefix = "w3")
```

__getitem__(*arg: Union[int, slice, str]*) → Union[Type[xarray.core.dataarray.DataArray], Type[xarray.core.dataset.Dataset]]

Get specified items from the disc.

Note: Make sure to have enough memory to load many files at the same time.

Parameters **arg** (int or slice or str) – Specifies the items to load from the disc, depending on the type of the argument:

- int returns the specified snapshot in a xarray.Dataset. It is equivalent to *Dataset.load_snapshot*;
- slice returns the specified snapshots in a xarray.Dataset;
- str returns the entire time series for a given variable in a xarray.DataArray. It is equivalent to *Dataset.load_time_series*;

Returns Xarray objects containing values loaded from the disc with the appropriate dimensions, coordinates and attributes.

Return type xarray.Dataset or xarray.DataArray

Raises TypeError – Raises type error if arg is not an interger, string or slice

Examples

Initial setup:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="input.i3d")
>>> prm.dataset.set(
...     filename_properties=dict(
...         separator="-",
...         file_extension=".bin",
...         number_of_digits=3
...     ),
...     drop_coords="z",
...     stack_scalar=True,
...     stack_velocity=True,
... )
```

- Load the entire time series for a given variable:

```
>>> ux = prm.dataset["ux"]
>>> uy = prm.dataset["uy"]
>>> uz = prm.dataset["uz"]
```

or organize them using a dataset:

```
>>> dataset = xarray.Dataset()
>>> for var in "ux uy uz".split():
...     dataset[var] = prm.dataset[var]
```

- Load all variables from a given snapshot:

```
>>> snapshot = prm.dataset[10]
```

- Load many snapshots at once with a `slice`, for instance, from 0 to 100 with a step of 10:

```
>>> snapshots = prm.dataset[0:101:10]
```

- Or simply load all snapshots at once (if you have enough memory):

```
>>> snapshots = prm.dataset[:]
```

__init__(***kwargs*)
Initializes the Dataset class.

Parameters

- **filename_properties** (*dict, optional*) – Keyword arguments for `FilenamePropertie`, like `separator`, `file_extension` and so on.
- ****kwargs** – Keyword arguments for the parameters, like `data_path`, `drop_coords` and so on.

Raises `KeyError` – Exception is raised when an Keyword arguments is not a valid parameter.

Returns An object to read and write the raw binary files from XCompact3d on-demand.

Return type *Dataset*

__iter__()
Yields all the snapshots, so the application can iterate over them.

Yields `xarray.Dataset` – Dataset containing the arrays loaded from the disc with the appropriate dimensions, coordinates and attributes.

Examples

Initial setup:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="input.i3d")
>>> prm.dataset.set(
...     filename_properties=dict(
...         separator="-",
...         file_extension=".bin",
...         number_of_digits=3
...     ),
...     stack_scalar=True,
...     stack_velocity=True,
... )
```

Iterate over all snapshots, loading them one by one:

```
>>> for ds in prm.dataset:
...     vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_derivative("y
↪")
...     prm.dataset.write(data = vort, file_prefix = "w3")
```

__len__() → int
Make the dataset work with the Python function `len`.

Returns Total of snapshots as a function of `snapshot_counting` and `snapshot_step`.

Return type int

__repr__()
Return repr(self).

load_array(*filename: str*, *add_time: bool = True*, *attrs: Optional[dict] = None*) → Type[xarray.core.dataarray.DataArray]
This method reads a binary field from XCompact3d with `numpy.fromfile` and wraps it into a `xarray.DataArray` with the appropriate dimensions, coordinates and attributes.

Parameters

- **filename** (*str*) – Name of the file.
- **add_time** (*bool, optional*) – Add time as a coordinate (default is `True`).
- **attrs** (*dict_like, optional*) – Attributes to assign to the new instance `xarray.DataArray`.

Returns Data array containing values loaded from the disc.

Return type `xarray.DataArray`

Examples

Initial setup:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="input.i3d")
>>> prm.dataset.set(
...     filename_properties=dict(
...         separator="-",
...         file_extension=".bin",
...         number_of_digits=3
...     ),
...     stack_scalar=True,
...     stack_velocity=True,
... )
```

Load one array from the disc:

```
>>> ux = prm.dataset.load_array("ux-000.bin")
```

load_snapshot(*numerical_identifier: int*, *list_of_variables: Optional[list] = None*, *add_time: bool = True*, *stack_scalar: Optional[bool] = None*, *stack_velocity: Optional[bool] = None*) → Type[xarray.core.dataset.Dataset]
Load the variables for a given snapshot.

Parameters

- **numerical_identifier** (*int*) – The number of the snapshot.
- **list_of_variables** (*list, optional*) – List of variables to be loaded, if None, it uses `Dataset.set_of_variables`, if `Dataset.set_of_variables` is empty, it automatically loads all arrays from this snapshot, by default None.
- **add_time** (*bool, optional*) – Add time as a coordinate, by default True.
- **stack_scalar** (*bool, optional*) – When true, the scalar fields will be stacked in a new coordinate `n`, otherwise returns one array per scalar fraction. If none, it uses `Dataset.stack_scalar`, by default None.

- **stack_velocity** (*bool, optional*) – When true, the velocity will be stacked in a new coordinate `i`, otherwise returns one array per velocity component. If none, it uses `Dataset.stack_velocity`, by default None.

Returns Dataset containing the arrays loaded from the disc with the appropriate dimensions, coordinates and attributes.

Return type `xarray.Dataset`

Raises `IOError` – Raises IO error if it does not find any variable for this snapshot.

Examples

Initial setup:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="input.i3d")
>>> prm.dataset.set(
...     filename_properties=dict(
...         separator="-",
...         file_extension=".bin",
...         number_of_digits=3
...     ),
...     stack_scalar=True,
...     stack_velocity=True,
... )
```

Load all variables from a given snapshot:

```
>>> snapshot = prm.dataset.load_snapshot(10)
```

or just

```
>>> snapshot = prm.dataset[10]
```

load_time_series(*array_prefix: str*) → Type[xarray.core.dataarray.DataArray]

Load the entire time series for a given variable.

Note: Make sure to have enough memory to load all files at the same time.

Parameters **array_prefix** (*str*) – Name of the variable, for instance `ux`, `uy`, `uz`, `pp`, `phi1`.

Returns DataArray containing the time series loaded from the disc, with the appropriate dimensions, coordinates and attributes.

Return type `xarray.DataArray`

Raises `IOError` – Raises IO error if it does not find any snapshot for this variable.

Examples

Initial setup:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="input.i3d")
>>> prm.dataset.set(
...     filename_properties=dict(
...         separator="-",
...         file_extension=".bin",
...         number_of_digits=3
...     ),
...     stack_scalar=True,
...     stack_velocity=True,
... )
```

Load the entire time series for a given variable:

```
>>> ux = prm.dataset.load_time_series("ux")
>>> uy = prm.dataset.load_time_series("uy")
>>> uz = prm.dataset.load_time_series("uz")
```

or just:

```
>>> ux = prm.dataset["ux"]
>>> uy = prm.dataset["uy"]
>>> uz = prm.dataset["uz"]
```

You can organize them using a dataset:

```
>>> dataset = xarray.Dataset()
>>> for var in "ux uy uz".split():
...     dataset[var] = prm.dataset[var]
```

load_wind_turbine_data (*file_pattern: Optional[str] = None*) → Type[xarray.core.dataset.Dataset]

Load the data produced by wind turbine simulations.

Note: This feature is experimental

Parameters **file_pattern** (*str, optional*) – A filename pattern used to locate the files with `glob.iglob`. If None, it is obtained from `datapath`, i.e., if `datapath = ./examples/Wind-Turbine/data` then `file_pattern = ./examples/Wind-Turbine/data/../*.perf`. By default None.

Returns A dataset with all variables as a function of the time

Return type `xarray.Dataset`

Examples

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="NREL-5MW.i3d")
>>> ds = prm.dataset.load_wind_turbine_data()
>>> ds
<xarray.Dataset>
Dimensions:           (t: 21)
Coordinates:
* t                  (t) float64 0.0 2.0 4.0 6.0 8.0 ... 34.0 36.0 38.0 40.0
Data variables: (12/14)
```

(continues on next page)

(continued from previous page)

```
    Number of Revs    (t) float64 0.0 0.4635 0.9281 1.347 ... 7.374 7.778 8.181
    GeneratorSpeed    (t) float64 0.0 149.3 133.6 123.2 ... 122.9 122.9 123.0
    GeneratorTorque   (t) float64 0.0 2.972e+04 3.83e+04 ... 4.31e+04 4.309e+04
    BladePitch1       (t) float64 0.0 12.0 14.21 13.21 ... 11.44 11.44 11.44
    BladePitch2       (t) float64 0.0 12.0 14.21 13.21 ... 11.44 11.44 11.44
    BladePitch3       (t) float64 0.0 12.0 14.21 13.21 ... 11.44 11.44 11.44
    ...                ...
    Ux                (t) float64 0.0 15.0 15.0 15.0 15.0 ... 15.0 15.0 15.0␣
↪15.0
    Uy                (t) float64 0.0 -1.562e-05 2.541e-05 ... 7.28e-07 7.683e-
↪07
    Uz                (t) float64 0.0 1.55e-06 ... -1.828e-06 2.721e-06
    Thrust            (t) float64 9.39e+05 1.826e+05 ... 4.084e+05 4.066e+05
    Torque            (t) float64 8.78e+06 1.268e+06 ... 4.231e+06 4.203e+06
    Power             (t) float64 1.112e+07 1.952e+06 ... 5.362e+06 5.328e+06
```

set (***kwargs*)

Set new values for any of the properties after the initialization.

Parameters

- **filename_properties** (*dict, optional*) – Keyword arguments for *FilenameProperties*, like `separator`, `file_extension` and so on.
- ****kwargs** – Keyword arguments for the parameters, like `data_path`, `drop_coords` and so on.

Raises `KeyError` – Exception is raised when an Keyword arguments is not a valid parameter.

Examples

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="input.i3d")
>>> prm.dataset.set(
...     filename_properties=dict(
...         separator="-",
...         file_extension=".bin",
...         number_of_digits=3
...     ),
...     stack_scalar=True,
...     stack_velocity=True,
... )
```

write (*data: Union[xarray.core.dataarray.DataArray, xarray.core.dataset.Dataset], file_prefix: Optional[str] = None*)

Write an array or dataset to raw binary files on the disc, in the same order that Xcompact3d would do, so they can be easily read with 2DECOMP.

In order to avoid overwriting any relevant field, only the variables in a dataset with an **attribute** called `file_name` will be written.

Coordinates are properly aligned before writing.

If `n` is a valid coordinate (for scalar fractions) in the array, one numerated binary file will be written for each scalar field.

If `i` is a valid coordinate (for velocity components) in the array, one binary file will be written for each of them (x, y or z).

If `t` is a valid coordinate (for time) in the array, one numerated binary file will be written for each available time.

Parameters

- **data** (`xarray.DataArray` or `xarray.Dataset`) – Data to be written
- **file_prefix** (*str, optional*) – filename prefix for the array, if data is `xarray.DataArray`, by default None

Raises `IOError` – Raises IO error data is not an `xarray.DataArray` or `xarray.Dataset`.

Examples

Initial setup:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="input.i3d")
>>> prm.dataset.set(
...     filename_properties=dict(
...         separator="-",
...         file_extension=".bin",
...         number_of_digits=3
...     ),
...     stack_scalar=True,
...     stack_velocity=True,
... )
```

- From a dataset, write only the variables with the atribute `file_name`,

notice that `ux` and `uy` will not be overwritten because them do not have the atribute `file_name`:

```
>>> for ds in prm.dataset:
...     ds["vort"] = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.  first_
↪derivative("y")
...     ds["vort"].attrs["file_name"] = "vorticity"
...     prm.dataset.write(ds)
```

- Write an array:

```
>>> for ds in prm.dataset:
...     vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_
↪derivative("y")
...     vort.attrs["file_name"] = "vorticity"
...     prm.dataset.write(vort)
```

or

```
>>> for ds in prm.dataset:
...     vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_
↪derivative("y")
...     prm.dataset.write(data = vort, file_prefix = "vorticity")
```

Note: It is not recomended to load the arrays with `add_time = False` when planning to write the results in a time series (e.g., *vort-000.bin*, *vort-001.bin*, *vort-002.bin*, ...)

write_xdmf (*xdmf_name: str = 'snapshots.xdmf'*) → None

Write the xdmf file, so the results from the simulation and its postprocessing can be opened in an external visualization tool, like Paraview.

Make sure to set all the parameters in this object properly.

If `set_of_objects` is empty, the files are obtained automatically with `glob.glob`.

Parameters **xdmf_name** (*str, optional*) – Filename for the xdmf file, by default "snapshots.xdmf"

Raises `IOError` – Raises IO error if it does not find any file for this simulation.

Examples

Initial setup:

```
>>> prm = xcompact3d_toolbox.Parameters(loadfile="input.i3d")
>>> prm.dataset.set(
...     filename_properties=dict(
...         separator="-",
...         file_extension=".bin",
...         number_of_digits=3
...     ),
...     stack_scalar=True,
...     stack_velocity=True,
... )
```

It is possible to produce a new xdmf file, so all data can be visualized on any external tool:

```
>>> prm.dataset.write_xdmf()
```

class xcompact3d_toolbox.io.**FilenameProperties**(***kwargs*)

Bases: `traitlets.traitlets.HasTraits`

Filename properties are important to guarantee consistency for input/output operations. This class makes xcompact3d-toolbox work with different types of file names for the binary fields produced from the numerical simulations and their pre/postprocessing.

Parameters

- **separator** (*str*) – The string used as separator between the name of the variable and its numeration, it can be an empty string (default is `"-"`).
- **file_extension** (*str*) – The file extension that identify the raw binary files from XCompact3d, it can be an empty string (default is `".bin"`).
- **number_of_digits** (*int*) – The number of numerical digits used to identify the time series (default is `3`).
- **scalar_num_of_digits** (*int*) – The number of numerical digits used to identify each scalar field (default is `1`).

Notes

FilenameProperties is in fact an atribute of *xcompact3d_toolbox.io.Dataset*, so there is no need to initialize it manually for most of the common use cases.

__init__(***kwargs*)
Initializes the object.

Parameters ****kwargs** – Keyword arguments for the parameters, like `separator`, `file_extension` and so on.

Raises `KeyError` – Raises an error if the user tries to set an invalid parameter.

Returns Filename properties

Return type *xcompact3d_toolbox.io.FilenameProperties*

__repr__()
Return repr(self).

get_filename_for_binary(*prefix: str*, *counter: int*, *data_path=''*) → str
Get the filename for an array.

Parameters

- **prefix** (*str*) – Name of the array.
- **counter** (*int or str*) – The number that identifies this array, it can be the string `"*"` if the filename is going to be used with `glob.glob`.
- **data_path** (*str*) – Path to the folder where the data is stored.

Returns The filename

Return type str

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.dataset.filename_properties.set(
...     separator = "-",
...     file_extension = ".bin",
...     number_of_digits = 3
... )
>>> prm.dataset.filename_properties.get_filename_for_binary("ux", 10)
'ux-010.bin'
>>> prm.dataset.filename_properties.get_filename_for_binary("ux", "*")
'ux-???.bin'
```

```
>>> prm.dataset.filename_properties.set(
...     separator = "",
...     file_extension = "",
...     number_of_digits = 4
... )
>>> prm.dataset.filename_properties.get_filename_for_binary("ux", 10)
'ux0010'
>>> prm.dataset.filename_properties.get_filename_for_binary("ux", "*")
'ux????'
```

get_info_from_filename(*filename: str*) → tuple[int, str]
Get information from the name of a binary file.

Parameters **filename** (*str*) – The name of the array.

Returns A tuple with the name of the array and the number that identifies it.

Return type tuple[int, str]

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.dataset.filename_properties.set(
...     separator = "-",
...     file_extension = ".bin",
...     number_of_digits = 3
... )
>>> prm.dataset.filename_properties.get_info_from_filename('ux-010.bin')
(10, 'ux')
```

```
>>> prm.dataset.filename_properties.set(
...     separator = "",
...     file_extension = "",
...     number_of_digits = 4
... )
>>> prm.dataset.filename_properties.get_info_from_filename("ux0010")
(10, 'ux')
```

get_name_from_filename (*filename: str*) → str
Same as *get_info_from_filename*, but just returns the name.

get_num_from_filename (*filename: str*) → int
Same as *get_info_from_filename*, but just returns the number.

set (***kwargs*) → None
Set new values for any of the properties after the initialization.

Parameters ****kwargs** – Keyword arguments for parameters, like `separator`, `file_extension` and so on.

Raises `KeyError` – Raises an error if the user tries to set an invalid parameter.

Examples

If the simulated fields are named like `ux-000.bin`, they are in the default configuration, there is no need to specify filename properties. But just in case, it would be like:

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.dataset.filename_properties.set(
...     separator = "-",
...     file_extension = ".bin",
...     number_of_digits = 3
... )
```

If the simulated fields are named like `ux0000`, the parameters are:

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> prm.dataset.filename_properties.set(
...     separator = "",
...     file_extension = "",
...     number_of_digits = 4
... )
```

4.1.5 Computation and Plotting

The data structure is provided by xarray, that introduces labels in the form of dimensions, coordinates and attributes on top of raw NumPy-like arrays, which allows for a more intuitive, more concise, and less error-prone developer experience.

See xarray's User Guide for a complete overview about its data structures and built-in functions for indexing, selecting, computing, plotting and much more. It integrates tightly with dask for parallel computing.

Consider using hvPlot to explore your data interactively, see how to plot Gridded Data.

Xcompact3d-toolbox adds extra functions on top of `xarray.DataArray` and `xarray.Dataset`, all the details are described bellow.

class xcompact3d_toolbox.array.**X3dDataArray**(*data_array*)

An accessor with extra utilities for `xarray.DataArray`.

cumtrapz(*dim*)

Cumulatively integrate `xarray.DataArray` in direction `dim` using the composite trapezoidal rule. It is a wrapper for `scipy.integrate.cumtrapz`. Initial value is defined to zero.

Parameters **dim** (*str*) – Coordinate used for the integration.

Returns Integrated

Return type `xarray.DataArray`

Examples

```
>>> da.x3d.cumtrapz('t')
```

first_derivative(*dim*)

Compute first derivative with the 4th order accurate centered scheme.

It is fully functional with all boundary conditions available on XCompact3d and stretched mesh in the vertical direction (y). The **atribute** `BC` is used to store Boundary Condition information in a dictionary (see examples), default is `ncl1 = ncln = 2` and `npaire = 1`.

Parameters **dim** (*str*) – Coordinate used for the derivative.

Returns **differentiated**

Return type `xarray.DataArray`

Examples

```
>>> da.attrs['BC'] = {
...     'x': {
...         'ncl1': 1,
...         'ncln': 1,
...         'npaire': 0
...     },
...     'y': {
...         'ncl1': 2,
...         'ncln': 1,
...         'npaire': 1,
...         'istret': 0,
...         'beta': 1.0
```

(continues on next page)

(continued from previous page)

```
...     },
...     'z': {
...         'ncl1': 0,
...         'ncln': 0,
...         'npaire': 1
... }
>>> da.x3d.first_derivative('x')
```

or just:

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> da.attrs['BC'] = prm.get_boundary_condition('ux')
>>> da.x3d.first_derivative('x')
```

pencil_decomp(**args*)

Coerce the data array into dask array.

It applies `chunk=-1` for all coordinates listed in `args`, which means no decomposition, and `'auto'` to the others, resulting in a pencil decomposition for parallel evaluation.

For customized `chunks` adjust, see `xarray.DataArray.chunk`.

Parameters **arg** (*str or sequence of str*) – Dimension(s) to apply no decomposition.

Returns **chunked**

Return type `xarray.DataArray`

Raises `ValueError` – args must be valid dimensions in the data array

Examples

```
>>> da.x3d.pencil_decomp('x') # Pencil decomposition
>>> da.x3d.pencil_decomp('t')
>>> da.x3d.pencil_decomp('y', 'z') # Slab decomposition
```

second_derivative(*dim*)

Compute second derivative with the 4th order accurate centered scheme.

It is fully functional with all boundary conditions available on Xcompact3d and stretched mesh in y direction. The **atribute** `BC` is used to store Boundary Condition information in a dictionary (see examples), default is `ncl1 = ncln = 2` and `npaire = 1`.

Parameters **dim** (*str*) – Coordinate used for the derivative.

Returns **differentiated**

Return type `xarray.DataArray`

Examples

```
>>> da.attrs['BC'] = {
...     'x': {
...         'ncl1': 1,
...         'ncln': 1,
...         'npaire': 0
```

(continues on next page)

(continued from previous page)

```
...     },
...     'y': {
...         'ncl1': 2,
...         'ncln': 1,
...         'npaire': 1
...         'istret': 0,
...         'beta': 1.0
...     },
...     'z': {
...         'ncl1': 0,
...         'ncln': 0,
...         'npaire': 1
... }
>>> da.x3d.second_derivative('x')
```

or just:

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> da.attrs['BC'] = prm.get_boundary_condition('ux')
>>> da.x3d.second_derivative('x')
```

simps(**args*)

Integrate `xarray.DataArray` in direction(s) `args` using the composite Simpson's rule. It is a wrapper for `scipy.integrate.simps`.

Parameters **arg** (*str or sequence of str*) – Dimension(s) to compute integration.

Returns Integrated

Return type `xarray.DataArray`

Raises `ValueError` – args must be valid dimensions in the data array

Examples

```
>>> da.x3d.simps('x')
>>> da.x3d.simps('t')
>>> da.x3d.simps('x', 'y', 'z')
```

class xcompact3d_toolbox.array.**X3dDataset**(*data_set*)

An accessor with extra utilities for `xarray.Dataset`.

cumtrapz(*dim*)

Cumulatively integrate all arrays in this dataset in direction `dim` using the composite trapezoidal rule. It is a wrapper for `scipy.integrate.cumtrapz`. Initial value is defined to zero.

Parameters **dim** (*str*) – Coordinate used for the integration.

Returns **Integrated**

Return type `xarray.Dataset`

Examples

```
>>> ds.x3d.cumtrapz('t')
```

pencil_decomp(**args*)

Coerce all arrays in this dataset into dask arrays.

It applies `chunk=-1` for all coordinates listed in `args`, which means no decomposition, and `"auto"` to the others, resulting in a pencil decomposition for parallel evaluation.

For customized `chunks` adjust, see `xarray.Dataset.chunk`.

Parameters **arg** (*str or sequence of str*) – Dimension(s) to apply no decomposition.

Returns **chunked**

Return type `xarray.Dataset`

Raises `ValueError` – args must be valid dimensions in the dataset

Examples

```
>>> ds.x3d.pencil_decomp('x') # Pencil decomposition
>>> ds.x3d.pencil_decomp('t')
>>> ds.x3d.pencil_decomp('y', 'z') # Slab decomposition
```

simps(**args*)

Integrate all arrays in this dataset in direction(s) `args` using the composite Simpson's rule. It is a wrapper for `scipy.integrate.simps`.

Parameters **arg** (*str or sequence of str*) – Dimension(s) to compute integration.

Returns **Integrated**

Return type `xarray.Dataset`

Raises `ValueError` – args must be valid dimensions in the dataset

Examples

```
>>> ds.x3d.simps('x')
>>> ds.x3d.simps('t')
>>> ds.x3d.simps('x', 'y', 'z')
```

4.1.6 Sandbox

The new **Sandbox Flow Configuration** (`itype = 12`) aims to break many of the barriers to entry in a Navier-Stokes solver. The idea is to easily provide everything that XCompact3d needs from a Python Jupyter Notebook, like initial conditions, solid geometry, boundary conditions, and the parameters. For students in computational fluid dynamics, it provides a direct hands-on experience and a safe place for practicing and learning, while for advanced users and code developers, it works as a rapid prototyping tool. For more details, see:

- "A Jupyter sandbox environment coupled into the high-order Navier-Stokes solver Xcompact3d", by F.N. Schuch, F.D. Vianna, A. Mombach, J.H. Silvestrini. JupyterCon 2020.
- "Sandbox flow configuration: A rapid prototyping tool inside XCompact3d", by F.N. Schuch. XCompact3d 2021 Online Showcase Event.

class xcompact3d_toolbox.sandbox.**Geometry**(*data_array*)

An accessor with some standard geometries for xarray.DataArray. Use them in combination with the arrays initialized at *xcompact3d_toolbox.sandbox.init_epsi* and the new *xcompact3d_toolbox.genepsi.gene_epsi_3D*.

ahmed_body(*scale=1.0*, *angle=45.0*, *wheels=False*, *remp=True*, ***kwargs*)

Draw an Ahmed body.

Parameters

- **scale** (*float*) – Ahmed body's scale (the default is 1).
- **angle** (*float*) – Ahmed body's angle at the back, in degrees (the default is 45).
- **wheel** (*bool*) – Draw "wheels" if True (the default is False).
- **remp** (*bool*) – Adds the geometry to the xarray.DataArray if True and removes it if False (the default is True).
- ****kwargs** (*float*) – Ahmed body's center.

Returns Array with(out) the Ahmed body

Return type xarray.DataArray

Raises

- KeyError – Center coordinates must be valid dimensions.
- NotImplementedError – Body must be centered in z.

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> epsi = xcompact3d_toolbox.init_epsi(prm)
>>> for key in epsi.keys():
>>>     epsi[key] = epsi[key].geo.ahmed_body(x=2)
```

box(*remp=True*, ***kwargs*)

Draw a box.

Parameters

- **remp** (*bool*) – Adds the geometry to the xarray.DataArray if True and removes it if False (the default is True).
- ****kwargs** (*tuple of float*) – Box's boundaries.

Returns Array with(out) the box

Return type xarray.DataArray

Raises KeyError – Boundaries coordinates must be valid dimensions

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> epsi = xcompact3d_toolbox.init_epsi(prm)
>>> for key in epsi.keys():
>>>     epsi[key] = epsi[key].geo.box(x=(2,5), y=(0,1))
```

cylinder(*radius=0.5*, *axis='z'*, *height=None*, *remp=True*, ***kwargs*)
Draw a cylinder.

Parameters

- **radius** (*float*) – Cylinder's radius (the default is 0.5).
- **axis** (*str*) – Cylinder's axis (the default is "z").
- **height** (*float or None*) – Cylinder's height (the default is None), if None, it will take the entire axis, otherwise $\pm h/2$ is considered from the center.
- **remp** (*bool*) – Adds the geometry to the xarray.DataArray if True and removes it if False (the default is True).
- ****kwargs** (*float*) – Cylinder's center point.

Returns Array with(out) the cylinder

Return type xarray.DataArray

Raises KeyError – Center coordinates must be valid dimensions

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> epsi = xcompact3d_toolbox.init_epsi(prm)
>>> for key in epsi.keys():
>>>     epsi[key] = epsi[key].geo.cylinder(x=4.0, y=5.0)
```

from_stl(*filename: str = None*, *stl_mesh: stl.mesh.Mesh = None*, *origin: dict = None*, *rotate: dict = None*, *scale: float = None*, *user_tol: float = 6.283185307179586*, *remp: bool = True*)
Load a STL file and compute if the nodes of the computational mesh are inside or outside the object. In this way, the customized geometry can be used at the flow solver.

The methodology is based on the work of:

- Jacobson, A., Kavan, L., & Sorkine-Hornung, O. (2013). Robust inside-outside segmentation using generalized winding numbers. ACM Transactions on Graphics (TOG), 32(4), 1-12.

The Python implementation is an adaptation from inside-3d-mesh (licensed under the MIT License), by @marmakoide.

To maximize the performance here at the toolbox, *from_stl* is powered by Numba, that translates Python functions to optimized machine code at runtime. This method is compatible with Dask for parallel computation. In addition, just the subdomain near the object is tested, to save computational time.

Note: The precision of the method is influenced by the complexity of the STL mesh, there is no guarantee it will work for all geometries. This feature is experimental, its interface may change in future releases.

Parameters

- **filename** (*str, optional*) – Filename of the STL file to be loaded and included in the cartesian domain, by default None
- **scale** (*float, optional*) – This parameters can be used to scale up the object when greater than one and scale it down when smaller than one, by default None

- **rotate** (*dict, optional*) – Rotate the object, including keyword arguments that are expected by `stl.mesh.Mesh.rotate`, like `axis`, `theta` and `point`. For more details, see numpy-stl's documentation. By default None
- **origin** (*dict, optional*) – Specify the location of the origin point for the geometry. It is considered as the minimum value computed from all points in the object for each coordinate, after scaling and rotating them. The keys of the dictionary are the coordinate names (`x`, `y` and `z`) and the values are the origin on that coordinate. For missing keys, the value is assumed as zero. By default None
- **stl_mesh** (*stl.mesh.Mesh, optional*) – For very customizable control over the 3D object, you can provide it directly. Note that none of the arguments above are applied in this case. For more details about how to create and modify the geometry, see numpy-stl's documentation. By default None
- **user_tol** (*float, optional*) – Control the tolerance used to compute if a mesh node is inside or outside the object. Values smaller than the default may reduce the number of false negatives. By default 2π
- **remp** (*bool, optional*) – Add the geometry to the `xarray.DataArray` if `True` and removes it if `False`, by default True

Returns Array with(out) the customized geometry

Return type `xarray.DataArray`

Raises

- `ValueError` – If neither `filename` or `stl_mesh` are specified
- `ValueError` – If `stl_mesh` is not valid, the test is performed by `stl.mesh.Mesh.check`
- `ValueError` – If `stl_mesh` is not closed, the test is performed by `stl.mesh.Mesh.is_closed`

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> epsi = xcompact3d_toolbox.init_epsi(prm, dask = True)
>>> for key in epsi.keys():
>>>     epsi[key] = epsi[key].geo.from_stl(
...         "My_file.stl",
...         scale=1.0,
...         rotate=dict(axis=[0, 0.5, 0], theta=math.radians(90)),
...         origin=dict(x=2, y=1, z=0),
...     )
```

mirror (*dim='x'*)

Mirror the ϵ array with respect to the central plane in the direction `dim`.

Parameters **dim** (*str*) – Reference for the mirror (the default is `x`).

Returns Mirrored array

Return type `xarray.DataArray`

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> epsi = xcompact3d_toolbox.init_epsi(prm)
>>> for key in epsi.keys():
>>>     epsi[key] = epsi[key].geo.cylinder(x=4, y=5).geo.mirror("x")
```

sphere (*radius=0.5*, *remp=True*, ***kwargs*)
Draw a sphere.

Parameters

- **radius** (*float*) – Sphere's radius (the default is 0.5).
- **remp** (*bool*) – Adds the geometry to the xarray.DataArray if True and removes it if False (the default is True).
- ****kwargs** (*float*) – Sphere's center.

Returns Array with(out) the sphere

Return type xarray.DataArray

Raises KeyError – Center coordinates must be valid dimensions

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> epsi = xcompact3d_toolbox.init_epsi(prm)
>>> for key in epsi.keys():
>>>     epsi[key] = epsi[key].geo.sphere(x=1, y=1, z=1)
```

square (*length=1.0*, *thickness=0.1*, *remp=True*, ***kwargs*)
Draw a squared frame.

Parameters

- **length** (*float*) – Frame's external length (the default is 1).
- **thickness** (*float*) – Frames's tickness (the default is 0.1).
- **remp** (*bool*) – Adds the geometry to the xarray.DataArray if True and removes it if False (the default is True).
- ****kwargs** (*float*) – Frames's center.

Returns Array with(out) the squared frame

Return type xarray.DataArray

Raises KeyError – Center coordinates must be valid dimensions

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> epsi = xcompact3d_toolbox.init_epsi(prm)
>>> for key in epsi.keys():
>>>     epsi[key] = epsi[key].geo.square(x=5, y=2, z=1)
```

xcompact3d_toolbox.sandbox.**init_dataset**(*prm*)

This function initializes a xarray.Dataset including all variables that should be provided to XCompact3d and the sandbox flow configuration, according to the computational and physical parameters.

Parameters **prm** (*xcompact3d_toolbox.parameters.Parameters*) – Contains the computational and physical parameters.

Returns

Each variable is initialized with np.zeros(dtype=xcompact3d_toolbox.param["mytype"]) and wrapped into a xarray.Dataset with the proper size, dimensions, coordinates and attributes, check them for more details. The variables are:

- bxx1, bxy1, bxz1 - Inflow boundary condition for ux, uy and uz, respectively (if nclx1 = 2);
- noise_mod_x1 - for random noise modulation at inflow boundary condition (if nclx1 = 2);
- bxphi1 - Inflow boundary condition for scalar field(s) (if nclx1 = 2 and numscalar > 0);
- byphi1 - Bottom boundary condition for scalar field(s) (if nclyS1 = 2, numscalar > 0 and uset = 0);
- byphin - Top boundary condition for scalar field(s) (if nclySn = 2, numscalar > 0 and uset = 0);
- ux, uy, uz - Initial condition for velocity field;
- phi - Initial condition for scalar field(s) (if numscalar > 0);
- vol_frc - Integral operator employed for flow rate control in case of periodicity in x direction (nclx1 = 0 and nclxn = 0). Xcompact3d will compute the volumetric integration as I = sum(vol_frc * ux) and them will correct streamwise velocity as ux = ux / I, so, set vol_frc properly.

Return type xarray.Dataset

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> dataset = xcompact3d_toolbox.init_dataset(prm)
>>> #
>>> # Code here your customized flow configuration
>>> #
>>> prm.dataset.write(dataset) # write the files to the disc
```

xcompact3d_toolbox.sandbox.**init_epsi**(*prm*, *dask=False*)

Initializes the ϵ arrays that define the solid geometry for the Immersed Boundary Method.

Parameters

- **prm** (*xcompact3d_toolbox.parameters.Parameters*) – Contains the computational and physical parameters.
- **dask** (*bool*) – Defines the lazy parallel execution with dask arrays. See xcompact3d_toolbox.array.x3d.pencil_decomp().

Returns

A dictionary containing the epsi(s) array(s):

- epsi (nx, ny, nz) if `iibm` != 0;
- xepsi (nxraf, ny, nz) if `iibm` = 2;
- yepsi (nx, nyraf, nz) if `iibm` = 2;
- zepsi (nx, ny, nzraf) if `iibm` = 2.

Each one initialized with np.zeros(dtype=np.bool) and wrapped into a `xarray.DataArray` with the proper size, dimensions and coordinates. They are used to define the object(s) that is(are) going to be inserted into the cartesian domain by the Immersed Boundary Method (IBM). They should be set to one (True) at the solid points and stay zero (False) at the fluid points, some standard geometries are provided by the accessor *xcompact3d_toolbox.sandbox.Geometry*.

Return type `dict` of `xarray.DataArray`

Examples

```
>>> prm = xcompact3d_toolbox.Parameters()
>>> epsi = xcompact3d_toolbox.init_epsi(prm)
```

4.1.7 Genepsi

This module generates all the files necessary for our customized Immersed Boundary Method, based on Lagrange reconstructions. It is an adaptation to Python from the original Fortran code and methods from:

- Gautier R., Laizet S. & Lamballais E., 2014, A DNS study of jet control with microjets using an alternating direction forcing strategy, Int. J. of Computational Fluid Dynamics, 28, 393–410.

gene_epsi_3D is powered by Numba, it translates Python functions to optimized machine code at runtime. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN.

`xcompact3d_toolbox.genepsi.`**`gene_epsi_3D`**(*epsi_in_dict*, *prm*)

This function generates all the auxiliar files necessary for our customize IBM, based on Lagrange reconstructions. The arrays can be initialized with *xcompact3d_toolbox.sandbox.init_epsi()*, then, some standard geometries are provided by the accessor *xcompact3d_toolbox.sandbox.Geometry*. Notice that you can apply our own routines for your own objects. The main outputs of the function are written to disc at the files `epsilon.bin`, `nobjx.dat`, `nobjy.dat`, `nobjz.dat`, `nxifpif.dat`, `nyifpif.dat`, `nzifpif.dat`, `xixf.dat`, `yiyf.dat` and `zizf.dat`. They will be used by Xcompact3d and the sandbox flow configuration.

Parameters

- **`epsi_in_dict`** (`dict` of `xarray.DataArray`) – A dictionary containing the epsi(s) array(s).
- **`prm`** (*xcompact3d_toolbox.parameters.Parameters*) – Contains the computational and physical parameters.

Returns All computed variables are returned in a Dataset if `prm.iibm >= 2`, but just for reference, since all the relevant values are written to the disc.

Return type `xarray.Dataset` or `None`

Examples

```
>>> prm = x3d.Parameters()
>>> epsi = x3d.sandbox.init_epsi(prm)
>>> for key in epsi.keys():
...     epsi[key] = epsi[key].geo.cylinder(x=4, y=5)
>>> dataset = x3d.gene_epsi_3D(epsi, prm)
```

Remember to set the number of objects after that if `prm.iibm >= 2`:

```
>>> if prm.iibm >= 2:
...     prm.nobjmax = dataset.obj.size
```

4.1.8 Sample Data

`xcompact3d_toolbox.tutorial.`**`open_dataset`**(*name: str*, ***kws*) → tuple[xr.Dataset, Parameters]

Open a dataset from the online repository (requires internet).

If a local copy is found then always use that to avoid network traffic.

Available datasets: * `"cylinder"`: Flow around a cylinder

Parameters

- **`name`** (*`str`*) – Name of the file containing the dataset. e.g. 'cylinder'.
- **`**kws`** (*`dict, optional`*) – Passed to `xarray.tutorial.open_dataset`

See also:

`xarray.open_dataset()`

4.2 Tutorials

4.2.1 Parameters

The computational and physical parameters are handled by `class xcompact3d_toolbox.Parameters`. It is built on top of Traitlets, which aims to make the parameters compatible with what xcompact3d expects, and also brings some advantages:

- Attributes are type-checked;
- Default values, restrictions and connections between related parameters are applied where necessary;
- 'On change' callbacks for validation and observation;
- Two-way linking with ipywidgets.

[1]:
```
import numpy as np
import xcompact3d_toolbox as x3d
```

The first step is to establish numerical precision. Use `np.float64` if Xcompact3d was compiled with the flag `-DDOUBLE_PREC` (check the Makefile), use `np.float32` otherwise:

[2]:
```
x3d.param["mytype"] = np.float32
```

4.2.1.1 Initialization

There are a few ways to initialize the class. First, calling it with no arguments initializes all variables with default value:

[3]:
```
prm = x3d.Parameters()
```

You can access a list with all the available variables at the Api reference.

Let's see how it looks like:

[4]:
```
print(prm)
```

```
! -*- mode: f90 -*-

!====================
&BasicParam
!====================

       C_filter = 0.49            !
           beta = 1.0             ! Refinement parameter
             dt = 0.001           ! Time step
          gravx = 0.0             ! Gravity unitary vector in x-direction
          gravy = 0.0             ! Gravity unitary vector in y-direction
          gravz = 0.0             ! Gravity unitary vector in z-direction
        ifilter = 0               !
         ifirst = 0               ! The number for the first iteration
           iibm = 0               ! Flag for immersed boundary method (0: No, 1: Yes)
            iin = 0               ! Defines perturbation at initial condition
          ilast = 0               ! The number for the last iteration
        ilesmod = 0               ! Enables Large-Eddy methodologies (0: No, 1: Yes)
           ilmn = .false.         !
   inflow_noise = 0.0             ! Turbulence intensity (1=100%) !! Inflow condition
     init_noise = 0.0             ! Turbulence intensity (1=100%) !! Initial condition
          ipost = 0               ! Enables online postprocessing at a frequency␣
↪iprocessing (0: No, 1: Yes)
        iscalar = 0               !
         istret = 0               ! y mesh refinement (0:no, 1:center, 2:both sides,␣
↪3:bottom)
       iturbine = 0               !
          itype = 12              ! Flow configuration (1:Lock-exchange, 2:TGV, 3:
↪Channel, and others)
          ivisu = 1               ! Enable store snapshots at a frequency ioutput (0:␣
↪No, 1: Yes)
          nclx1 = 2               ! Velocity boundary condition where x=0
          nclxn = 2               ! Velocity boundary condition where x=xlx
          ncly1 = 2               ! Velocity boundary condition where y=0
          nclyn = 2               ! Velocity boundary condition where y=yly
          nclz1 = 2               ! Velocity boundary condition where z=0
          nclzn = 2               ! Velocity boundary condition where z=zlz
      numscalar = 0               ! Number of scalar fractions
             nx = 17              ! X-direction nodes
             ny = 17              ! Y-direction nodes
             nz = 17              ! Z-direction nodes
          p_col = 0               ! Column partition for domain decomposition and␣
↪parallel computation
          p_row = 0               ! Row partition for domain decomposition and␣
↪parallel computation
             re = 1000.0          ! Reynolds number
```

(continues on next page)

(continued from previous page)

```
             u1 = 2.0              !
             u2 = 1.0              !
            xlx = 1.0              ! Size of the box in x-direction
            yly = 1.0              ! Size of the box in y-direction
            zlz = 1.0              ! Size of the box in z-direction

/End

!====================
&NumOptions
!====================

            cnu = 0.44             ! Ratio between hyperviscosity at km=2/3 and kc=␣
↪(dissipation factor range)
      ifirstder = 4                !
      iimplicit = 0                !
     isecondder = 4                ! Scheme for second order derivative
    itimescheme = 3                ! Time integration scheme (1: Euler, 2: AB2, 3: AB3,
↪ 5: RK3)
          nu0nu = 4.0              ! Ratio between hyperviscosity/viscosity at nu␣
↪(dissipation factor intensity)

/End

!====================
&InOutParam
!====================

    icheckpoint = 1000             ! Frequency for writing backup file
       ioutflow = 0                !
        ioutput = 1000             ! Frequency for visualization file
    iprocessing = 1000             ! Frequency for online postprocessing
       irestart = 0                ! Read initial flow field (0: No, 1: Yes)
       ninflows = 1                !
        nprobes = 0                !
     ntimesteps = 1                !
          nvisu = 1                ! Size for visualization collection
       output2D = 0                !

/End

!====================
&Statistics
!====================


/End

!====================
&CASE
!====================


/End
```

It is possible to access and/or set values afterwards:

[5]:
```
# Reynolds Number
print(prm.re)

# attribute new value
prm.re = 1e6
print(prm.re)
```

```
1000.0
1000000.0
```

Second, we can specify some values, and let the missing ones be initialized with default value:

[6]:
```
prm = x3d.Parameters(
    filename="example.i3d",
    itype=10,
    nx=129,
    ny=65,
    nz=32,
    xlx=15.0,
    yly=10.0,
    zlz=3.0,
    nclx1=2,
    nclxn=2,
    ncly1=1,
    nclyn=1,
    nclz1=0,
    nclzn=0,
    iin=1,
    istret=2,
    re=300.0,
    init_noise=0.0125,
    inflow_noise=0.0125,
    dt=0.0025,
    ifirst=1,
    ilast=45000,
    irestart=0,
    icheckpoint=45000,
    ioutput=200,
    iprocessing=50,
)
```

It is easy to write `example.i3d` to disc, just type:

[7]:
```
prm.write()
```

And finally, it is possible to read the parameters from the disc:

[8]:
```
prm = x3d.Parameters(filename="example.i3d")
prm.load()
```

The same result is obtained in a more concise way:

[9]:
```
prm = x3d.Parameters(loadfile="example.i3d")
```

The class can also read the previous parameters format (se more information here):

```
prm = x3d.Parameters(loadfile="incompact3d.prm")
```

There are extra objects read and write the raw binary files from XCompact3d on-demand.

- Read a binary field from the disc:

```
ux = prm.dataset.load_array("ux-0000.bin")
```

- Read the entire time series for a given variable:

```
ux = prm.dataset.load_time_series("ux")
```

- Read all variables for a given snapshot:

```
snapshot = prm.dataset.load_snapshot(10)
```

- Write `xdmf` files, so the binary files can be open in any external visualization tool:

```
prm.dataset.write_xdmf()
```

- Compute the coordinates, including support for mesh refinement in y:

[10]:
```
prm.get_mesh()
```

[10]:
```
{'x': array([ 0.       ,  0.1171875,  0.234375 ,  0.3515625,  0.46875  ,
         0.5859375,  0.703125 ,  0.8203125,  0.9375   ,  1.0546875,
         1.171875 ,  1.2890625,  1.40625  ,  1.5234375,  1.640625 ,
         1.7578125,  1.875    ,  1.9921875,  2.109375 ,  2.2265625,
         2.34375  ,  2.4609375,  2.578125 ,  2.6953125,  2.8125   ,
         2.9296875,  3.046875 ,  3.1640625,  3.28125  ,  3.3984375,
         3.515625 ,  3.6328125,  3.75     ,  3.8671875,  3.984375 ,
         4.1015625,  4.21875  ,  4.3359375,  4.453125 ,  4.5703125,
         4.6875   ,  4.8046875,  4.921875 ,  5.0390625,  5.15625  ,
         5.2734375,  5.390625 ,  5.5078125,  5.625    ,  5.7421875,
         5.859375 ,  5.9765625,  6.09375  ,  6.2109375,  6.328125 ,
         6.4453125,  6.5625   ,  6.6796875,  6.796875 ,  6.9140625,
         7.03125  ,  7.1484375,  7.265625 ,  7.3828125,  7.5      ,
         7.6171875,  7.734375 ,  7.8515625,  7.96875  ,  8.0859375,
         8.203125 ,  8.3203125,  8.4375   ,  8.5546875,  8.671875 ,
         8.7890625,  8.90625  ,  9.0234375,  9.140625 ,  9.2578125,
         9.375    ,  9.4921875,  9.609375 ,  9.7265625,  9.84375  ,
         9.9609375, 10.078125 , 10.1953125, 10.3125   , 10.4296875,
        10.546875 , 10.6640625, 10.78125  , 10.8984375, 11.015625 ,
        11.1328125, 11.25     , 11.3671875, 11.484375 , 11.6015625,
        11.71875  , 11.8359375, 11.953125 , 12.0703125, 12.1875   ,
        12.3046875, 12.421875 , 12.5390625, 12.65625  , 12.7734375,
        12.890625 , 13.0078125, 13.125    , 13.2421875, 13.359375 ,
        13.4765625, 13.59375  , 13.7109375, 13.828125 , 13.9453125,
        14.0625   , 14.1796875, 14.296875 , 14.4140625, 14.53125  ,
        14.6484375, 14.765625 , 14.8828125, 15.       ], dtype=float32),
 'y': array([ 0.        ,  0.04504663,  0.09029302,  0.13594234,  0.18220465,
         0.2293007 ,  0.27746594,  0.3269554 ,  0.37804887,  0.43105724,
         0.4863303 ,  0.5442656 ,  0.60532016,  0.670024  ,  0.7389978 ,
         0.81297535,  0.8928308 ,  0.9796148 ,  1.0746001 ,  1.1793398 ,
         1.2957417 ,  1.4261622 ,  1.5735214 ,  1.7414377 ,  1.9343702 ,
         2.1577313 ,  2.4178853 ,  2.72186   ,  3.0764673 ,  3.4864438 ,
         3.9514096 ,  4.462363  ,  5.        ,  5.537637  ,  6.0485907 ,
         6.5135565 ,  6.923533  ,  7.27814   ,  7.5821147 ,  7.842269  ,
         8.06563   ,  8.258562  ,  8.426478  ,  8.573838  ,  8.704258  ,
         8.820661  ,  8.9254    ,  9.020385  ,  9.107169  ,  9.187025  ,
         9.261003  ,  9.329976  ,  9.39468   ,  9.455734  ,  9.51367   ,
         9.568943  ,  9.621951  ,  9.673044  ,  9.722534  ,  9.7706995 ,
```

(continues on next page)

(continued from previous page)

```
        9.817796  ,  9.864058  ,  9.909707  ,  9.954953  , 10.        ],
      dtype=float32),
'z': array([0.     , 0.09375, 0.1875 , 0.28125, 0.375  , 0.46875, 0.5625 ,
       0.65625, 0.75   , 0.84375, 0.9375 , 1.03125, 1.125  , 1.21875,
       1.3125 , 1.40625, 1.5    , 1.59375, 1.6875 , 1.78125, 1.875  ,
       1.96875, 2.0625 , 2.15625, 2.25   , 2.34375, 2.4375 , 2.53125,
       2.625  , 2.71875, 2.8125 , 2.90625], dtype=float32)}
```

More details about I/O and array manipulations with xarray will be included in a new tutorial.

4.2.1.2 Traitlets

Type-checking

All parameters are type-checked, to make sure that they are what `Xcompact3d` expects. Use the cellcode below to see how a `TraitError` pops out when we try:

```
prm.itype = 10.5
prm.itype = -5
prm.itype = 20
prm.itype = 'sandbox'
```

[ ]:

Validation

Some parameters, like mesh points (`nx`, `ny` and `nz`), trigger a validation operation when a new value is attributed to them. Due to restrictions at the FFT library, they must be equal to:

$$n_i = \begin{cases} 2^{1+a} \times 3^b \times 5^c & \text{if periodic,} \\ 2^{1+a} \times 3^b \times 5^c + 1 & \text{otherwise,} \end{cases}$$

where a, b and c are non negative integers. In addition, the derivatives stencil imposes that:

$$n_i \geq \begin{cases} 8 & \text{if periodic,} \\ 9 & \text{otherwise.} \end{cases}$$

Again, give it a try at the cellcode below:

```
prm.nx = 129
prm.nx = 4
prm.nx = 60
prm.nx = 61
```

[ ]:

Observation

Other parameters, like mesh resolution (`dx`, `dy` and `dz`), are automatically updated when any new attribution occurs to mesh points and/or domain size. Let's create a quick print functions to play with:

[11]:
```
def show_param():
    for var in "nclx1 nclxn nx xlx dx".split():
        print(f"{var:>5} = {getattr(prm, var)}")
```

We are starting with:

[12]:
```
show_param()
```

```
nclx1 = 2
nclxn = 2
   nx = 129
  xlx = 15.0
   dx = 0.1171875
```

Let's change just the domain's length:

[13]:
```
prm.xlx = 50.0

show_param()
```

```
nclx1 = 2
nclxn = 2
   nx = 129
  xlx = 50.0
   dx = 0.390625
```

The resolution was updated as well. Now the number of mesh points:

[14]:
```
prm.nx = 121

show_param()
```

```
nclx1 = 2
nclxn = 2
   nx = 121
  xlx = 50.0
   dx = 0.4166666666666667
```

Again, the resolution was updated. Now we set a new mesh resolution, this time, `xlx` will be updated in order to satisfy the new resolution:

[15]:
```
prm.dx = 1e-2

show_param()
```

```
nclx1 = 2
nclxn = 2
   nx = 121
  xlx = 1.2
   dx = 0.01
```

Boundary conditions are observed as well. Xcompact3d allows three different BC for velocity:

- Periodic `0`;
- Free-slip `1`;
- Dirichlet `2`.

They can be assigned individually for each of the six boundaries:

- `nclx1` and `nclxn`, where $x = 0$ and $x = xlx$;

- `ncly1` and `nclyn`, where $y = 0$ and $y = yly$;
- `nclz1` and `nclzn`, where $z = 0$ and $z = zlz$.

It leads to 5 possibilities (`00`, `11`, `12`, `21` and `22`), because both boundary must be periodic, or not, so `0` cannot be combined.

Let's check it out, we are starting with:

[16]:
```
show_param()
```

```
nclx1 = 2
nclxn = 2
   nx = 121
  xlx = 1.2
   dx = 0.01
```

We will change just one side to periodic (`nclx1 = 0`), for consistence, the other side should be periodic too. Let's see:

[17]:
```
prm.nclx1 = 0

show_param()
```

```
nclx1 = 0
nclxn = 0
   nx = 120
  xlx = 1.2
   dx = 0.01
```

Now free-slip in one side (`nclx1 = 1`), and the other should be non-periodic:

[18]:
```
prm.nclx1 = 1

show_param()
```

```
nclx1 = 1
nclxn = 1
   nx = 121
  xlx = 1.2
   dx = 0.01
```

Setting the other boundary to periodic:

[19]:
```
prm.nclxn = 0

show_param()
```

```
nclx1 = 0
nclxn = 0
   nx = 120
  xlx = 1.2
   dx = 0.01
```

and now back to Dirichlet:

[20]:
```
prm.nclxn = 2

show_param()
```

```
nclx1 = 2
nclxn = 2
   nx = 121
  xlx = 1.2
   dx = 0.01
```

This time, free-slip:

[21]:
```
prm.nclxn = 1

show_param()
```

```
nclx1 = 2
nclxn = 1
   nx = 121
  xlx = 1.2
   dx = 0.01
```

There was no need to update `nclx1`, because `1` and `2` can be freely combined. Notice that `nx` was modified properly from 121 to 120 and then back, according to the possible values, `dx` and `xlx` stayed untouched.

Metadata

Traitlets types constructors have a `tag` method to store metadata in a dictionary. In the case of Xcompact3d-toolbox, two are especially useful:

- `group` defines to what namespace a given parameter belongs when the class is written to `.i3d` file (`.write()` method) or read from `.i3d` or `.prm` files (`.load()` method), parameters without a group are ignored for both methods;
- `desc` contains a brief description of each parameter that is shown on screen as we saw above, and also printed with the `.write()` method.

Declaring new parameters

You probably would like to add more parameters for your own flow configuration, or because some of them were not implemented yet (it is a work in progress).

To do so, any auxiliar variable can be included after initialization, like:

[22]:
```
prm.my_variable = 0  # or any other datatype
```

It was called auxiliar variable because, in this way, it will be available only for the Python application.

In order to include it at the `.i3d` file and make it available for XCompact3d, we can create a subclass that inherits all the functionality from `xcompact3d_toolbox.Parameters`:

[23]:
```
import traitlets


# Create a class named my_Parameters, which inherits the properties all properties␣
↪and methods
class my_Parameters(x3d.Parameters):
    # .tag with group and description guarantees that the new variable will
    # be compatible with all functionalities (like .write() and .load())
    a_my_variable = traitlets.Int(default_value=0, min=0).tag(
```

(continues on next page)

(continued from previous page)

```
        group="BasicParam", desc="An example at the Tutorial <------"
    )

    # And a custom method, for instance
    def my_method(self):
        return self.a_my_variable * 2


prm = my_Parameters(nx=257, ny=129, nz=31, a_my_variable=10)  # and here we go

# Testing the method
print(prm.my_method())

# Show all parameters on screen
print(prm)
```

```
20
! -*- mode: f90 -*-

!====================
&BasicParam
!====================

       C_filter = 0.49            !
  a_my_variable = 10              ! An example at the Tutorial <------
           beta = 1.0             ! Refinement parameter
             dt = 0.001           ! Time step
          gravx = 0.0             ! Gravity unitary vector in x-direction
          gravy = 0.0             ! Gravity unitary vector in y-direction
          gravz = 0.0             ! Gravity unitary vector in z-direction
        ifilter = 0               !
         ifirst = 0               ! The number for the first iteration
           iibm = 0               ! Flag for immersed boundary method (0: No, 1: Yes)
            iin = 0               ! Defines perturbation at initial condition
          ilast = 0               ! The number for the last iteration
        ilesmod = 0               ! Enables Large-Eddy methodologies (0: No, 1: Yes)
           ilmn = .false.         !
   inflow_noise = 0.0             ! Turbulence intensity (1=100%) !! Inflow condition
     init_noise = 0.0             ! Turbulence intensity (1=100%) !! Initial condition
          ipost = 0               ! Enables online postprocessing at a frequency␣
↪iprocessing (0: No, 1: Yes)
        iscalar = 0               !
         istret = 0               ! y mesh refinement (0:no, 1:center, 2:both sides,␣
↪3:bottom)
       iturbine = 0               !
          itype = 12              ! Flow configuration (1:Lock-exchange, 2:TGV, 3:
↪Channel, and others)
          ivisu = 1               ! Enable store snapshots at a frequency ioutput (0:␣
↪No, 1: Yes)
          nclx1 = 2               ! Velocity boundary condition where x=0
          nclxn = 2               ! Velocity boundary condition where x=xlx
          ncly1 = 2               ! Velocity boundary condition where y=0
          nclyn = 2               ! Velocity boundary condition where y=yly
          nclz1 = 2               ! Velocity boundary condition where z=0
          nclzn = 2               ! Velocity boundary condition where z=zlz
     numscalar = 0                ! Number of scalar fractions
            nx = 257              ! X-direction nodes
```

(continues on next page)

(continued from previous page)

```
             ny = 129              ! Y-direction nodes
             nz = 31               ! Z-direction nodes
          p_col = 0                ! Column partition for domain decomposition and␣
↪parallel computation
          p_row = 0                ! Row partition for domain decomposition and␣
↪parallel computation
             re = 1000.0           ! Reynolds number
             u1 = 2.0              !
             u2 = 1.0              !
            xlx = 1.0              ! Size of the box in x-direction
            yly = 1.0              ! Size of the box in y-direction
            zlz = 1.0              ! Size of the box in z-direction

/End

!====================
&NumOptions
!====================

            cnu = 0.44             ! Ratio between hyperviscosity at km=2/3 and kc=␣
↪(dissipation factor range)
      ifirstder = 4                !
      iimplicit = 0                !
     isecondder = 4                ! Scheme for second order derivative
    itimescheme = 3                ! Time integration scheme (1: Euler, 2: AB2, 3: AB3,
↪ 5: RK3)
          nu0nu = 4.0              ! Ratio between hyperviscosity/viscosity at nu␣
↪(dissipation factor intensity)

/End

!====================
&InOutParam
!====================

    icheckpoint = 1000             ! Frequency for writing backup file
       ioutflow = 0                !
        ioutput = 1000             ! Frequency for visualization file
    iprocessing = 1000             ! Frequency for online postprocessing
       irestart = 0                ! Read initial flow field (0: No, 1: Yes)
       ninflows = 1                !
        nprobes = 0                !
     ntimesteps = 1                !
           nvisu = 1               ! Size for visualization collection
       output2D = 0                !

/End

!====================
&Statistics
!====================


/End

!====================
&CASE
```

(continues on next page)

(continued from previous page)

```
!====================


/End
```

Take a look at the source code of parameters.py if you need more examples for different datatypes.

4.2.1.3 Graphical User Interface

For an interactive experience launch this tutorial on Binder, the widgets are not so responsive when disconnected from a Python application.

To conclude this part of the tutorial, let's see another option to handle the parameters. The class `ParametersGui` is a subclass of `Parameters`, and includes all the features described above. In addition, `ParametersGui` offers an user interface with IPywidgets.

It is still under development, more parameters and features are going to be included soon, as well as more widgets.

Just like before, we start with:

[24]:
```
prm = x3d.ParametersGui()
```

Widgets are returned on demand when any instance of `class ParametersGui` is called, let's see:

[25]:
```
prm("nx", "xlx", "dx", "nclx1", "nclxn")
```

```
VBox(children=(Dropdown(description='nx', description_tooltip='X-direction nodes',␣
↪index=3, options=(9, 11, 13...
```

You can play around with the widgets above and see the effect of the observations made previously.

Notice that the Traitlets parameters are related to the value at their widgets in a **two-way link**, in this way, a print will show the actual value on the widgets:

[26]:
```
show_param()
```

```
nclx1 = 2
nclxn = 2
   nx = 17
  xlx = 1.0
   dx = 0.0625
```

Give it a try, modify the values at the widgets and print them again.

It also works on the other way, set a new value to a parameters will change its widget, try it:

[27]:
```
#prm.nclx1 = 0
```

And of course, different widgets for the same parameter are always synchronized, change the widget below and see what happens with the widget above:

[28]:
```
prm('nx')
```

```
VBox(children=(Dropdown(description='nx', description_tooltip='X-direction nodes',␣
↪index=3, options=(9, 11, 13...
```

A last example is about the domain decomposition for parallel computation, Xcompact3d uses 2DECOMP&FFT. The available options for `p_row` and `p_col` are presented as functions of the number of computational cores `ncores`, notice that `p_row * p_col = ncores` should be respected and `p_row * p_col = 0` activates the auto-tunning mode. The widgets are prepared to respect these restrictions:

[29]:
```
prm('ncores', 'p_row', 'p_col')
```

```
VBox(children=(BoundedIntText(value=4, description='ncores', max=1000000000),␣
↪Dropdown(description='p_row', de...
```

To conclude this part of the tutorial, let's see what happens when `class ParametersGui` is presented on screen, hover the mouse over some variable to see its description:

[30]:
```
prm
```

```
VBox(children=(HTML(value='<h1>Xcompact3d Parameters</h1>'),␣
↪HBox(children=(Text(value='input.i3d', descriptio...
```

4.2.2 Reading and writing files

This tutorial includes an overview of the different ways available to load the binary arrays from the disc after running a numerical simulation with XCompact3d. Besides that, some options are presented to save the results from our analysis, together with some tips and tricks.

For an interactive experience launch this tutorial on Binder.

4.2.2.1 Preparation

Here we prepare the dataset for this notebook, so it can be reproduced on local machines or on the cloud, you are invited to test and interact with many of the concepts. It also provides nice support for courses and tutorials, let us know if you produce any of them.

The very first step is to import the toolbox and other packages:

[1]:
```
import warnings

import numpy as np
import xarray as xr
import xcompact3d_toolbox as x3d
```

Then we can download an example from the online database, the flow around a cylinder in this case. We set `cache=True` and a local destination where it can be saved in our computer `cache_dir="./example/"`, so there is no need to download it everytime the kernel is restarted.

[2]:
```
cylinder_ds, prm = x3d.tutorial.open_dataset(
    "cylinder", cache=True, cache_dir="./example/"
)
```

let's take a look at the dataset:

[3]:
```
cylinder_ds.info()
```

```
xarray.Dataset {
dimensions:
        i = 2 ;
        x = 257 ;
        y = 128 ;
        t = 201 ;

variables:
        float32 u(i, x, y, t) ;
        float32 pp(x, y, t) ;
        float32 epsi(x, y) ;
        float64 x(x) ;
        float64 y(y) ;
        float64 t(t) ;
        object i(i) ;

// global attributes:
        :xcompact3d_version = v3.0-397-gff531df ;
        :xcompact3d_toolbox_version = 1.0.1 ;
        :url = https://github.com/fschuch/xcompact3d_toolbox_data ;
        :dataset_license = MIT License ;
}
```

We got a xarray.Dataset with the variables `u` (velocity vector), `pp` (pressure) and `epsi` (describes the geometry), their coordinates (`x`, `y`, `t` and `i`) and some atributes like the `xcompact3d_version` used to run this simulation, the `url` where you can find the dataset, and others.

In the next block, we configure the toolbox and some atributes at the dataset, so we can write all the binary fields to the disc. Do not worry about the details right now, this is just the preparation step, we are going to discuss them later.

[4]:
```
x3d.param["mytype"] = np.float32

prm.dataset.set(data_path="./data/", drop_coords="z")

cylinder_ds.u.attrs["file_name"] = "u"
cylinder_ds.pp.attrs["file_name"] = "pp"
cylinder_ds.epsi.attrs["file_name"] = "epsilon"

prm.write("input.i3d")

prm.dataset.write(cylinder_ds)

prm.dataset.write_xdmf("xy-planes.xdmf")

del cylinder_ds, prm
```

```
ux:   0%|          | 0/201 [00:00<?, ?it/s]
uy:   0%|          | 0/201 [00:00<?, ?it/s]
pp:   0%|          | 0/201 [00:00<?, ?it/s]
xy-planes.xdmf:   0%|          | 0/201 [00:00<?, ?it/s]
```

After that, the files are organized as follow:

```
tutorial
|   computing_and_plotting.ipynb
|   io.ipynb
```

(continues on next page)

(continued from previous page)

```
    input.i3d
    parameters.ipynb
    xy-planes.xdmf

└── data
        epsilon.bin
        pp-000.bin
        pp-001.bin
        ...
        pp-199.bin
        pp-200.bin
        ux-000.bin
        ux-001.bin
        ...
        ux-199.bin
        ux-200.bin
        uy-000.bin
        uy-001.bin
        ...
        uy-199.bin
        uy-200.bin
        uz-000.bin
        uz-001.bin
        ...
        uz-199.bin
        uz-200.bin

└── example
        cylinder.nc
```

It is very similar to what we get after successfully running a simulation, so now we can move on to the tutorial.

4.2.2.2 Why xarray?

The data structures are provided by xarray, that introduces labels in the form of dimensions, coordinates and attributes on top of raw NumPy-like arrays, which allows for a more intuitive, more concise, and less error-prone developer experience. It integrates tightly with dask for parallel computing.

The goal here is to speed up the development of customized post-processing applications with the concise interface provided by xarray. Ultimately, we can compute solutions with fewer lines of code and better readability, so we expend less time testing and debugging and more time exploring our datasets and getting insights.

Additionally, xcompact3d-toolbox includes extra functionalities for DataArray and Dataset.

Before going forward, please, take a look at Overview: Why xarray? and Quick overview to understand the motivation to use xarray's data structures instead of just numpy-like arrays.

4.2.2.3 Xarray objects on demand

To start our post-processing, let's load the parameters file:

[5]:
```
prm = x3d.Parameters(loadfile="input.i3d")
```

Notice there is an entire tutorial dedicated to it.

To save space on the disc, our dataset was converted from double precision to single, so we have to configure the toolbox to:

[6]:
```
x3d.param["mytype"] = np.float32
```

The methods in the toolbox support different filename properties, like the classic `ux000` or the new `ux-0000.bin`, besides some combinations between them. For our case, we set the parameters as:

[7]:
```
prm.dataset.filename_properties.set(
    separator = "-",
    file_extension = ".bin",
    number_of_digits = 3,
)
```

Now we specify the parameters for our dataset, like where it is found (`data_path`), if it needs to drop some coordinate (`drop_coords`, again, to save space, we are working with a span-wise averaged dataset, so we drop `z` to work with `xy` planes), we inform the parameter that controls the number of timesteps `snapshot_counting` and their step `snapshot_step`. Consult the dataset documentation to see different ways to customize your experience, and choose the ones that best suits your post-processing application. In this example, they are defined as:

[8]:
```
prm.dataset.set(
    data_path="./data/",
    drop_coords="z",
    snapshot_counting="ilast",
    snapshot_step="ioutput"
)
```

Now we are good to go.

We can check the length of the dataset we are dealing with:

[9]:
```
len(prm.dataset)
```

[9]:
```
201
```

Meaning that our binary files range from 0 (i.e., `ux-000.bin`) to 200 (i.e., `ux-200.bin`), exactly as expected.

It is possible to load any given array:

[10]:
```
epsilon = prm.dataset.load_array("./data/epsilon.bin", add_time=False)
```

Notice that load_array requires the entire path to the file, and we use `add_time=False` because this array does not evolve in time like the others, i.e., it is not numerated for several snapshots.

We can see it on the screen:

[11]:
```
epsilon
```

[11]:
```
<xarray.DataArray (x: 257, y: 128)>
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
```

Let's do it again, this time for `ux` and using `add_time=True`:

[12]:
```
ux = prm.dataset.load_array("./data/ux-100.bin", add_time=True)
```

See that `t` is now a coordinate, and for this snapshot it was computed automatically as dimensionless time `75.0`:

[13]:
```
ux
```

[13]:
```
<xarray.DataArray 'ux' (x: 257, y: 128, t: 1)>
array([[[1.        ],
        [1.        ],
        [1.        ],
        ...,
        [1.        ],
        [1.        ],
        [1.        ]],

       [[1.0000466 ],
        [0.99996716],
        [1.0000466 ],
        ...,
        [0.9999681 ],
        [1.0000459 ],
        [0.9999675 ]],

       [[1.0000602 ],
        [1.0000144 ],
        [1.0000602 ],
        ...,
...
        ...,
        [1.0140737 ],
        [1.0142432 ],
        [1.0144366 ]],

       [[1.0146521 ],
        [1.0148891 ],
        [1.0151588 ],
        ...,
        [1.0141058 ],
        [1.0142633 ],
        [1.0144445 ]],

       [[1.0146475 ],
        [1.0148702 ],
        [1.0151254 ],
        ...,
        [1.014144  ],
        [1.0142874 ],
        [1.014454  ]]], dtype=float32)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float32 75.0
```

That is not all. If you have enough memory, you can load the entire time series for a given variable with load_time_series, or simply by:

[14]:
```
ux = prm.dataset["ux"]
```
```
./data/ux-???.bin:   0%|          | 0/201 [00:00<?, ?it/s]
```

Let's see it (note 201 files are loaded and wrapped with the appropriate coordinates):

[15]:
```
ux
```
[15]:
```
<xarray.DataArray 'ux' (x: 257, y: 128, t: 201)>
array([[[0.9999885 , 1.        , 1.        , ..., 1.        ,
         1.        , 1.        ],
        [0.9999782 , 1.        , 1.        , ..., 1.        ,
         1.        , 1.        ],
        [0.9999907 , 1.        , 1.        , ..., 1.        ,
         1.        , 1.        ],
        ...,
        [1.0000122 , 1.        , 1.        , ..., 1.        ,
         1.        , 1.        ],
        [0.99999744, 1.        , 1.        , ..., 1.        ,
         1.        , 1.        ],
        [0.9999945 , 1.        , 1.        , ..., 1.        ,
         1.        , 1.        ]],

       [[0.99999535, 1.0000458 , 1.0000486 , ..., 1.0000464 ,
         1.0000468 , 1.0000476 ],
        [1.0000107 , 0.99997103, 0.99996907, ..., 0.9999675 ,
         0.99996674, 0.9999665 ],
        [1.0000069 , 1.0000455 , 1.0000486 , ..., 1.0000459 ,
         1.0000468 , 1.0000478 ],
...
        [0.9999908 , 1.0001078 , 1.000192  , ..., 1.0155317 ,
         1.0153327 , 1.0146557 ],
        [0.9999987 , 1.0001054 , 1.0001911 , ..., 1.0152053 ,
         1.0150691 , 1.0146141 ],
        [0.9999874 , 1.0000997 , 1.0001901 , ..., 1.0149139 ,
         1.0148364 , 1.0145978 ]],

       [[1.0000043 , 1.0000877 , 1.0001792 , ..., 1.0146514 ,
         1.0146394 , 1.0146087 ],
        [1.0000119 , 1.0000889 , 1.0001761 , ..., 1.014433  ,
         1.0144511 , 1.0146273 ],
        [1.0000004 , 1.0000932 , 1.000174  , ..., 1.0142416 ,
         1.0142918 , 1.0146769 ],
        ...,
        [1.0000123 , 1.0000978 , 1.0001832 , ..., 1.0154953 ,
         1.0153852 , 1.0147215 ],
        [0.99998474, 1.000096  , 1.0001816 , ..., 1.0151795 ,
         1.0151051 , 1.0146575 ],
        [0.99999154, 1.0000918 , 1.0001806 , ..., 1.0148991 ,
         1.0148568 , 1.0146192 ]]], dtype=float32)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float64 0.0 0.75 1.5 2.25 3.0 ... 147.0 147.8 148.5 149.2 150.0
```

You can store each array in a different variable, like:

[16]:
```
ux = prm.dataset["ux"]
```

(continues on next page)

(continued from previous page)

```
uy = prm.dataset["uy"]
pp = prm.dataset["pp"]
```

```
./data/ux-???.bin:   0%|          | 0/201 [00:00<?, ?it/s]
./data/uy-???.bin:   0%|          | 0/201 [00:00<?, ?it/s]
./data/pp-???.bin:   0%|          | 0/201 [00:00<?, ?it/s]
```

Or organize many arrays in a dataset:

[17]:
```
# create an empty dataset
ds = xr.Dataset()

# populate it
for var in ["ux", "uy", "pp"]:
    ds[var] = prm.dataset[var]

# show on the screen
ds
```

```
./data/ux-???.bin:   0%|          | 0/201 [00:00<?, ?it/s]
./data/uy-???.bin:   0%|          | 0/201 [00:00<?, ?it/s]
./data/pp-???.bin:   0%|          | 0/201 [00:00<?, ?it/s]
```

[17]:
```
<xarray.Dataset>
Dimensions:  (x: 257, y: 128, t: 201)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float64 0.0 0.75 1.5 2.25 3.0 ... 147.0 147.8 148.5 149.2 150.0
Data variables:
    ux       (x, y, t) float32 1.0 1.0 1.0 1.0 1.0 ... 1.015 1.015 1.015 1.015
    uy       (x, y, t) float32 9.98e-06 8.496e-08 ... -0.0005357 0.003209
    pp       (x, y, t) float32 0.0 0.03264 0.03613 ... 0.04078 0.03922 0.03859
```

It is possible to load all the variables from a given snapshot with load_snapshot, or simply:

[18]:
```
snapshot = prm.dataset[100]
```

And we got a xarray.Dataset with all the variables and their coordinates. You can access each of them with the dot notation (i.e., `snapshot.pp`, `snapshot.ux`, `snapshot.uy`) or the dict-like notation (i.e., `snapshot["pp"]`, `snapshot["ux"]`, `snapshot["uy"]`). See the dataset:

[19]:
```
snapshot
```

[19]:
```
<xarray.Dataset>
Dimensions:  (x: 257, y: 128, t: 1)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float32 75.0
Data variables:
    pp       (x, y, t) float32 0.05232 0.05219 0.05243 ... 0.03986 0.03989
    ux       (x, y, t) float32 1.0 1.0 1.0 1.0 1.0 ... 1.014 1.014 1.014 1.014
    uy       (x, y, t) float32 3.407e-07 1.503e-07 ... 0.007724 0.007703
```

Do you need the snapshots in a range? No problem. Let's do a slice to load the last 100, and just to exemplify, compute a time average:

[20]:
```
time_averaged = prm.dataset[-100:].mean("t")
time_averaged
```

[20]:
```
<xarray.Dataset>
Dimensions:  (x: 257, y: 128)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
Data variables:
    pp       (x, y) float32 0.05351 0.05335 0.05356 ... 0.03887 0.03886 0.03886
    ux       (x, y) float32 1.0 1.0 1.0 1.0 1.0 ... 1.015 1.015 1.015 1.015
    uy       (x, y) float32 -6.206e-09 2.081e-09 ... -6.504e-05 -6.531e-05
```

You can even use the slice notation to load all the snapshots at once:

[21]:
```
prm.dataset[:]
```

[21]:
```
<xarray.Dataset>
Dimensions:  (x: 257, y: 128, t: 201)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float64 0.0 0.75 1.5 2.25 3.0 ... 147.0 147.8 148.5 149.2 150.0
Data variables:
    pp       (x, y, t) float32 0.0 0.03264 0.03613 ... 0.04078 0.03922 0.03859
    ux       (x, y, t) float32 1.0 1.0 1.0 1.0 1.0 ... 1.015 1.015 1.015 1.015
    uy       (x, y, t) float32 9.98e-06 8.496e-08 ... -0.0005357 0.003209
```

Of course, some simulations may not fit in the memory like in this tutorial. For these cases we can iterate over all snapshots, loading them one by one:

[22]:
```
for ds in prm.dataset:
    # Computing the vorticity, just to exemplify
    vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_derivative("y")
```

Note that `reversed(prm.dataset)` also works.

Or for better control, we can iterate over a selected range of snapshots loading them one by one. The arguments are the same of a classic range in Python:

[23]:
```
for ds in prm.dataset(100, 200, 1):
    # Computing the vorticity, just to exemplify
    vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_derivative("y")
```

[24]:
```
# Result from the last iteration
vort
```

[24]:
```
<xarray.DataArray (y: 128, t: 1, x: 257)>
array([[[-0.00268999,  0.00181677,  0.0001196 , ...,  0.00324146,
          0.00302246, -0.01646465]],

       [[-0.0034743 , -0.00406752, -0.00344934, ...,  0.0029434 ,
          0.00254392, -0.01612649]],

       [[-0.00303078, -0.00340789, -0.0030852 , ...,  0.0026302 ,
          0.0024012 , -0.01718407]],

       ...,
```

(continues on next page)

(continued from previous page)

```
       [[-0.00178402, -0.00190772, -0.00161242, ...,  0.00415556,
          0.00375753, -0.01497194]],

       [[-0.00232817, -0.00317189, -0.00255065, ...,  0.00385816,
          0.00362989, -0.01590095]],

       [[-0.00263563,  0.00207518,  0.00038911, ...,  0.00355052,
          0.00316822, -0.01547501]]], dtype=float32)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float32 149.2
```

4.2.2.4 Writting the results to binary files

In the last example we computed the vorticity but did nothing with it. This time, let's write it to the disc using write:

[25]:
```
for ds in prm.dataset:
    vort = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_derivative("y")
    prm.dataset.write(data = vort, file_prefix = "w3")
```

The example above works for a xarray.DataArray. We can do it for a xarray.Dataset as well, but with one key difference. Only the arrays with an attribute called `file_name` will be written. It is done to avoid overwriting the base fields (`ux`, `uy`, `uz`, ...) by accident.

Let's rewrite the previous example to store `vort` in the dataset `ds`. We set an atribute `file_name` to `w3`, so the arrays will be written as `w3-000.bin`, `w3-001.bin`, `w3-002.bin`, etc.

We are also suppressing warnings, because the application will tell us it can not save `pp`, `ux` and `uy`, since they do not have a `file_name`. But in fact, we do not want to rewrite them anyway.

See the code:

[26]:
```
with warnings.catch_warnings():
    warnings.filterwarnings('ignore', category=UserWarning)
    for ds in prm.dataset:
        ds["vort"] = ds.uy.x3d.first_derivative("x") - ds.ux.x3d.first_derivative("y")
        ds["vort"].attrs["file_name"] = "w3"
        prm.dataset.write(ds)
```

The method prm.dataset.write() writes the files as raw binaries in the same way that XCompact3d would do. It means you can read them at the flow solver and also process them on any other tool that you are already familiar with, including the toolbox.

For instance, we get `w3` if we load snapshot 0 again:

[27]:
```
prm.dataset[0]
```

[27]:
```
<xarray.Dataset>
Dimensions:  (x: 257, y: 128, t: 1)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float32 0.0
Data variables:
```

(continues on next page)

(continued from previous page)

```
    pp       (x, y, t) float32 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    ux       (x, y, t) float32 1.0 1.0 1.0 1.0 1.0 1.0 ... 1.0 1.0 1.0 1.0 1.0
    uy       (x, y, t) float32 9.98e-06 1.955e-05 ... -2.285e-05 1.962e-05
    w3       (x, y, t) float32 0.00038 -0.000863 ... -0.001097 4.864e-05
```

Update the xdmf file

After computing and writing new results to the disc, you can open them on any external tools, like Paraview or Visit. You can update the xdmf file to include the recently computed `w3`. See the code:

[28]:
```
prm.dataset.write_xdmf("xy-planes.xdmf")
```

```
xy-planes.xdmf:   0%|          | 0/201 [00:00<?, ?it/s]
```

4.2.2.5 Other formats

Xarray objects can be exported to many other formats, depending on your needs.

For instance, xarray.DataArray and xarray.Dataset can be written as netCDF. In this way, they will keep all dimensions, coordinates, and attributes. This format is easier to handle and share because the files are self-sufficient. It is the format used to download the dataset used in this tutorial, and it is a good alternative to use when sharing the results of your research.

Just to give you an estimation about the disk usage, the size of the dataset `cylinder.nc` that we downloaded for this tutorial is 75.8 MB. The size of the folder `./data/` after producing the binary files in the same way that XCompact3d would do is 75.7 MB.

To exemplify the use of netCDF, let's take one snapshot:

[29]:
```
snapshot = prm.dataset[0]
snapshot
```

[29]:
```
<xarray.Dataset>
Dimensions:  (x: 257, y: 128, t: 1)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float32 0.0
Data variables:
    pp       (x, y, t) float32 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    ux       (x, y, t) float32 1.0 1.0 1.0 1.0 1.0 1.0 ... 1.0 1.0 1.0 1.0 1.0
    uy       (x, y, t) float32 9.98e-06 1.955e-05 ... -2.285e-05 1.962e-05
    w3       (x, y, t) float32 0.00038 -0.000863 ... -0.001097 4.864e-05
```

Now, let's include additional information for the ones that are going to use our data. You can set attributes for each array, coordinate, and also global attributes for the dataset. They are stored in a dictionary.

See the example:

[30]:
```
# Setting attributes for each coordinate
snapshot.x.attrs = dict(
    name = "x",
    long_name = "Stream-wise coordinate",
    units = "-"
)
```

(continues on next page)

(continued from previous page)

```
snapshot.y.attrs = dict(
    name = "y",
    long_name = "Vertical coordinate",
    units = "-"
)
snapshot.t.attrs = dict(
    name = "t",
    long_name = "Time",
    units = "-"
)
# Setting attributes for each array
snapshot.ux.attrs = dict(
    name = "ux",
    long_name = "Stream-wise velocity",
    units = "-"
)
snapshot.uy.attrs = dict(
    name = "y",
    long_name = "Vertical velocity",
    units = "-"
)
snapshot.pp.attrs = dict(
    name = "p",
    long_name = "Pressure",
    units = "-"
)
snapshot.w3.attrs = dict(
    name = "w3",
    long_name = "Vorticity",
    units = "-"
)
# Setting attributes for the dataset
snapshot.attrs = dict(
    title = "An example from the tutorials",
    url = "https://xcompact3d-toolbox.readthedocs.io/en/stable/tutorial/io.html",
    authors = "List of names",
    doi = "maybe a fancy doi from zenodo",
)
```

Exporting it as a netCDF file:

[31]:
```
snapshot.to_netcdf("snapshot-000.nc")
```

Importing the netCDF file:

[32]:
```
snapshot_in = xr.open_dataset("snapshot-000.nc")
```

See the result, it keeps all dimensions, coordinates, and attributes:

[33]:
```
snapshot_in
```

[33]:
```
<xarray.Dataset>
Dimensions:  (x: 257, y: 128, t: 1)
Coordinates:
  * x        (x) float32 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float32 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float32 0.0
```

(continues on next page)

(continued from previous page)

```
Data variables:
    pp       (x, y, t) float32 ...
    ux       (x, y, t) float32 ...
    uy       (x, y, t) float32 ...
    w3       (x, y, t) float32 ...
Attributes:
    title:    An example from the tutorials
    url:      https://xcompact3d-toolbox.readthedocs.io/en/stable/tutorial/io...
    authors:  List of names
    doi:      maybe a fancy doi from zenodo
```

We can compare them and see that their data, dimensions and coordinates are exactly the same:

[34]:
```
xr.testing.assert_equal(snapshot, snapshot_in)
```

Xarray is built on top of Numpy, so you can access a `numpy.ndarray` object with the property `data`. It is compatible with `numpy.save` and many other methods, see the example:

[35]:
```
np.save("epsi.npy", epsilon.data)
epsi_in = np.load("epsi.npy")

print(type(epsi_in))
epsi_in
```

```
<class 'numpy.ndarray'>
```

[35]:
```
array([[[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

You can use it for backwards compatibility with your previous post-processing tools. It is just not so effective, because we lost track of metadata like the coordinates and attributes.

If you manage to reduce the dataset's dimensions with some integral, average, or selecting subsets of data, you can convert it to a `pandas.Dataframe` and then export it to CSV, Excel, and many other options.

For instance, let's select a vertical profile for all variables where `x = 20` and convert it to a dataframe:

[36]:
```
snapshot_in.sel(x=20.0).to_dataframe()
```

[36]:

		x	pp	ux	uy	w3
y	t					
0.00000	0.0	20.0	0.0	1.000004	0.000011	-0.000180
0.09375	0.0	20.0	0.0	1.000012	-0.000005	-0.000235
0.18750	0.0	20.0	0.0	1.000000	0.000024	-0.000465
0.28125	0.0	20.0	0.0	1.000002	-0.000019	-0.000969
0.37500	0.0	20.0	0.0	0.999997	-0.000018	0.002280
...		...	...	...	...	...
11.53125	0.0	20.0	0.0	1.000013	0.000023	0.000101
11.62500	0.0	20.0	0.0	1.000009	-0.000010	-0.002189
11.71875	0.0	20.0	0.0	1.000012	-0.000005	0.000339
11.81250	0.0	20.0	0.0	0.999985	-0.000023	-0.001097
11.90625	0.0	20.0	0.0	0.999992	0.000020	0.000049

[128 rows x 5 columns]

Now, you can refer to pandas documentation for more details.

4.2.3 Computing and Plotting

This tutorial includes an overview of the different ways available to compute, select data and plot using the xarray objects that are provided by xcompact3d-toolbox.

The very first step is to import the toolbox and other packages:

[1]:
```
import hvplot.xarray
import matplotlib.pyplot as plt
import numpy as np
import xcompact3d_toolbox as x3d
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript

4.2.3.1 Why xarray?

The data structures are provided by xarray, that introduces labels in the form of dimensions, coordinates and attributes on top of raw NumPy-like arrays, which allows for a more intuitive, more concise, and less error-prone developer experience. It integrates tightly with dask for parallel computing.

The goal here is to speed up the development of customized post-processing applications with the concise interface provided by xarray. Ultimately, we can compute solutions with fewer lines of code and better readability, so we expend less time testing and debugging and more time exploring our datasets and getting insights.

Additionally, xcompact3d-toolbox includes extra functionalities for DataArray and Dataset.

Before going forward, please, take a look at Overview: Why xarray? and Quick overview to understand the motivation to use xarray's data structures instead of just numpy-like arrays.

4.2.3.2 Example - Flow around a cylinder

We can download the example from the online database, the flow around a cylinder in this case. We set `cache=True` and a local destination where it can be saved in our computer `cache_dir="./example/"`, so there is no need to download it every time the kernel is restarted.

[2]:
```
dataset, prm = x3d.tutorial.open_dataset("cylinder", cache=True, cache_dir="./example/
↪")
```

Notice there is an entire tutorial dedicated to the parameters file. Now, let's take a look at the dataset:

[3]:
```
dataset
```

[3]:
```
<xarray.Dataset>
Dimensions:  (i: 2, x: 257, y: 128, t: 201)
Coordinates:
  * x        (x) float64 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float64 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
```

(continues on next page)

(continued from previous page)

```
  * t        (t) float64 0.0 0.75 1.5 2.25 3.0 ... 147.0 147.8 148.5 149.2 150.0
  * i        (i) object 'x' 'y'
Data variables:
    u        (i, x, y, t) float32 ...
    pp       (x, y, t) float32 ...
    epsi     (x, y) float32 ...
Attributes:
    xcompact3d_version:          v3.0-397-gff531df
    xcompact3d_toolbox_version:  1.0.1
    url:                         https://github.com/fschuch/xcompact3d_toolbo...
    dataset_license:             MIT License
```

We got a xarray.Dataset with the variables `u` (velocity vector), `pp` (pressure) and `epsi` (that describes the geometry), their coordinates (`x`, `y`, `t` and `i`) and some atributes like the `xcompact3d_version` used to run this simulation, the `url` where you can find the dataset and others.

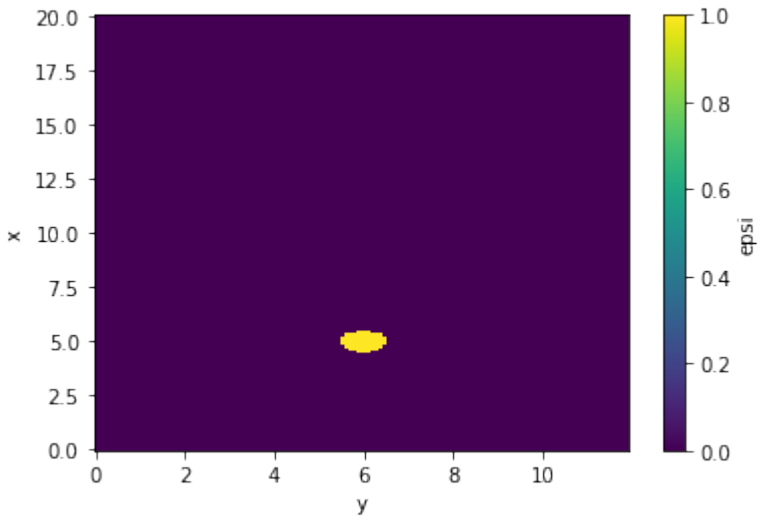
We can access each of variables or coordinates with the dot notation (i.e., `snapshot.pp`, `snapshot.u`, `snapshot.x`) or the dict-like notation (i.e., `snapshot["pp"]`, `snapshot["u"]`, `snapshot["x"]`).

Once the arrays are wrapped with their coordinates, we can use xarray's plotting functionality to explore our data with just a few lines of code.

Starting with `epsi`, that represents the geometry (it is 1 inside the cylinder and 0 outside), we select it from the dataset and then use the method plot:

[4]:
```
dataset.epsi.plot()
```

[4]:
```
<matplotlib.collections.QuadMesh at 0x7f50b8e7f610>
```

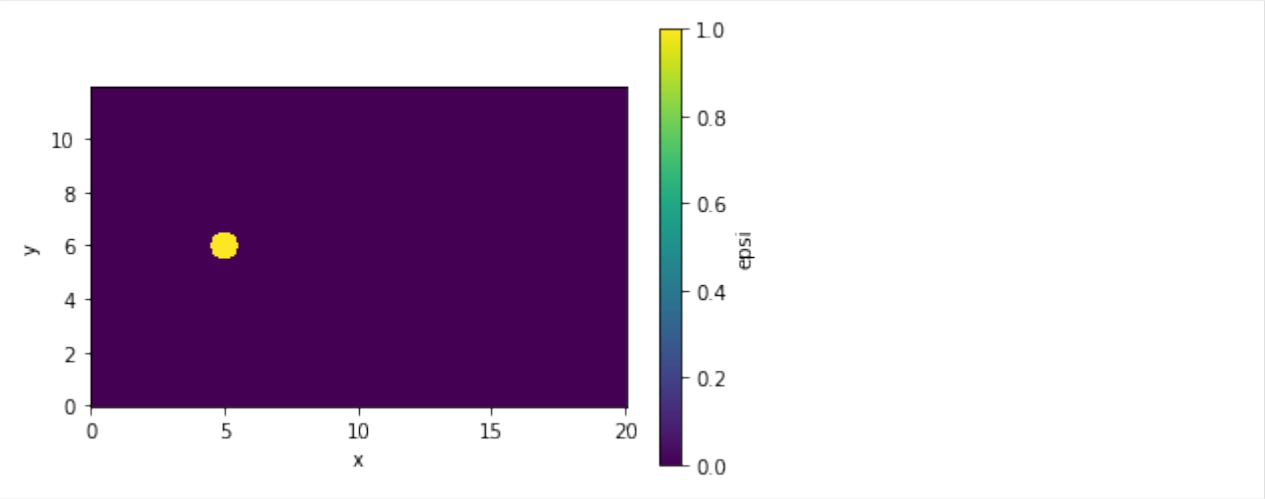


The array in the example was two-dimensional, in this case `.plot()` automatically calls xarray.plot.pcolormesh().

There are many options to customize the plots, besides that, xarray plotting functionality is a thin wrapper around the popular matplotlib library.

To improve the figure, let's set the x-axis of the plot as the coordinate `x` of our array, same for `y`. Then let's use matplotlib to set the axis aspect to `equal`. Take a look:

[5]:
```
ax = dataset.epsi.plot(x="x", y="y")
ax.axes.set_aspect("equal")
```



It might be important in the cylinder case to ignore the values that are inside the solid cylinder when plotting or computing any quantity. We can do it by preserving the values of `u` and `pp` where where `epsi` is equal to zero, and setting the values to `np.NaN` otherwise.

xarray.Dataset.where is a handy method for that, take a look:

[6]:
```
for var in ["u", "pp"]:
    dataset[var] = dataset[var].where(dataset.epsi == 0.0, np.NaN)
```

Have you noticed that we are doing this comparison between variables with different dimensions, and it just worked? I mean, `epsi` is 2D (x, y), `pp` is 3D (x, y, t) and `u` is 4D (i, x, y, t). That is because xarray automatically broadcasted the values of `epsi` to each point at the coordinates `t` and `i`.

Another cool feature of xarray is that we can select data based on the actual value of its coordinates, not only on the integer indexes used for selection on numpy-like arrays.

To exemplify, let's select one position at the same heigh of the cylinder, but a bit downstream. Note we can get the time evolution for all variables at this specified point:

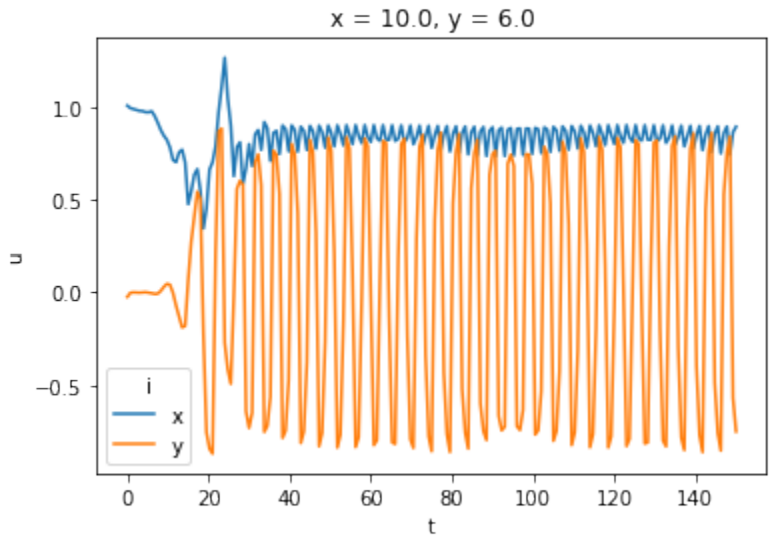
[7]:
```
dataset.sel(x=10.0, y=6.0, method="nearest")
```

[7]:
```
<xarray.Dataset>
Dimensions:  (i: 2, t: 201)
Coordinates:
    x        float64 10.0
    y        float64 6.0
  * t        (t) float64 0.0 0.75 1.5 2.25 3.0 ... 147.0 147.8 148.5 149.2 150.0
  * i        (i) object 'x' 'y'
Data variables:
    u        (i, t) float32 1.005 0.9915 0.987 0.9834 ... 0.838 -0.5676 -0.749
    pp       (t) float32 0.0 -0.003727 -0.001281 ... -0.4308 -0.5678 -0.2354
    epsi     float32 0.0
Attributes:
    xcompact3d_version:          v3.0-397-gff531df
    xcompact3d_toolbox_version:  1.0.1
    url:                         https://github.com/fschuch/xcompact3d_toolbo...
    dataset_license:             MIT License
```

We can chain the methods, selecting a variable, selecting a point in the domain and doing a plot, all with just one line of code:

[8]:
```
dataset.u.sel(x=10.0, y=6.0, method="nearest").plot(x="t", hue="i")
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7f50b8b9e410>,
 <matplotlib.lines.Line2D at 0x7f50b8c0e790>]
```

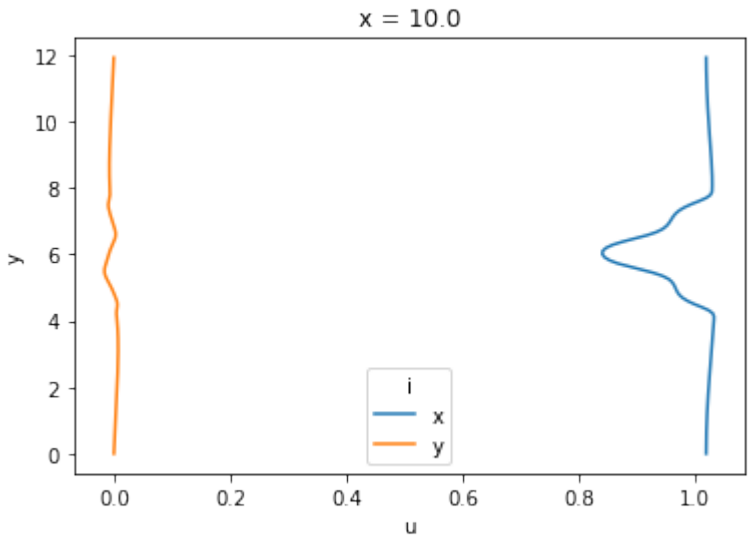


Note this time the data was 1D, so the plot was handled internally by xarray.plot.line.

To give you another example, let's plot the time-averaged ($60 \leq t \leq 150$) vertical velocity profile where $x = 10$:

[9]:
```
dataset.u.sel(x=10.0, t=slice(60.0, 150.0)).mean("t").plot(y="y", hue="i")
```

[9]:
```
[<matplotlib.lines.Line2D at 0x7f50b8b425d0>,
 <matplotlib.lines.Line2D at 0x7f50b8b37710>]
```



As you saw, we can refer to the coordinates by their name when working with xarray, instead of keeping track of their axis number.

To extend this concept, let's now compute the time evolution of the kinetic energy in our flow. It is given by the

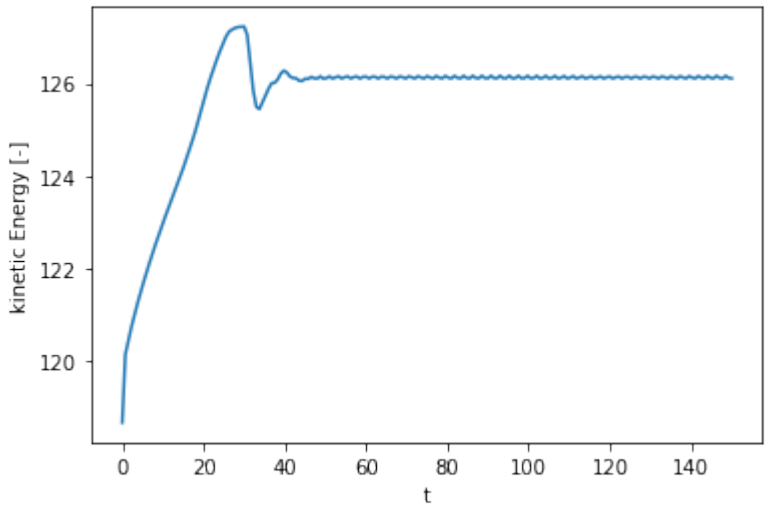
equation:

$$k = \int_V \frac{u_i u_i}{2} dV.$$

Now the code:

[10]:
```
dataset["kinetic_energy"] = ((dataset.u ** 2.0).sum("i").x3d.simps("x", "y")) * 0.5
dataset["kinetic_energy"].attrs = dict(name="k", long_name="kinetic Energy", units="-
↪")
dataset["kinetic_energy"].plot()
```

[10]:
```
[<matplotlib.lines.Line2D at 0x7f50b8a621d0>]
```



In the code above we:

- Solved the equation with a very readable code. A good point is that it worked for the `xy` planes in the dataset in this example, and all we need to do to run it in a real 3D case is include `z` at the integration;
- Included attributes to describe what we just computed, making our application easier to share and collaborate. As a bonus, they were automatically included in the plot;
- Plotted the results.

We can use a quick list comprehension to get the dimensions for the volumetric integration:

[11]:
```
V_coords = [dim for dim in dataset.u.coords if dim in "xyz"]
V_coords
```

[11]:
```
['x', 'y']
```

and rewrite the previous example to make it more robust, now it works for 1D, 2D and 3D cases:

[12]:
```
dataset["kinetic_energy"] = ((dataset.u ** 2.0).sum("i").x3d.simps(*V_coords)) * 0.5
dataset["kinetic_energy"].attrs = dict(name="k", long_name="kinetic Energy", units="-
↪")
```

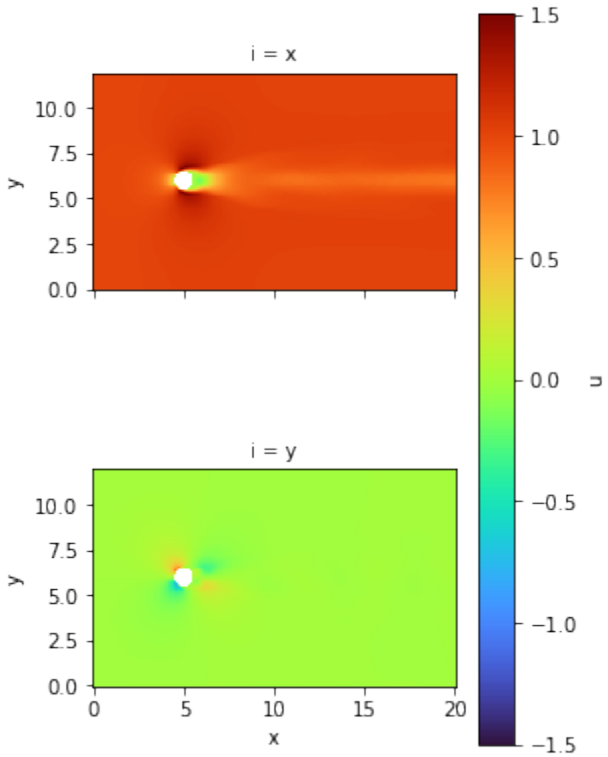
Going back to 2D plots, let's take the velocity vector `u`, select it for $60 \leq t \leq 150$ (sel), compute a time average (mean) and plot:

[13]:
```
g = dataset.u.sel(t=slice(60.0, 150.0)).mean("t").plot(
    x="x", y="y", row="i", cmap="turbo", rasterized=True
```

(continues on next page)

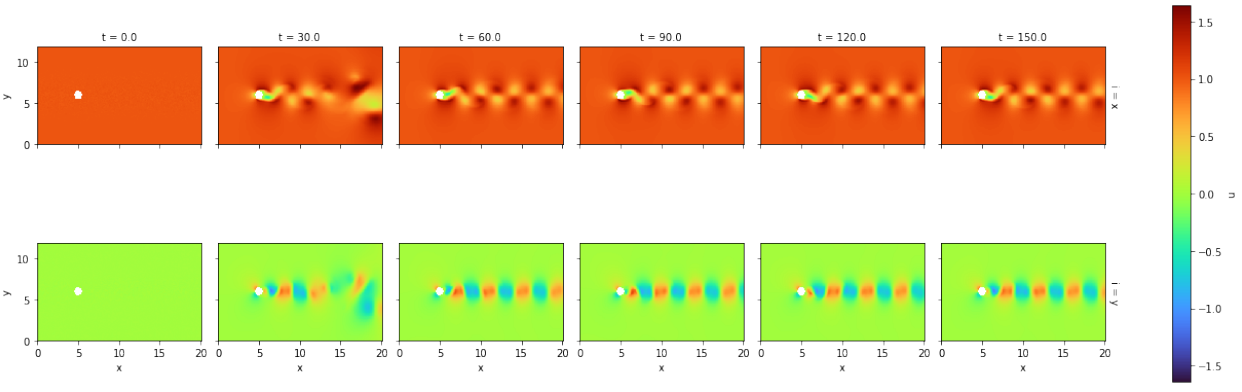
(continued from previous page)

```
)
for ax in g.axes.flat:
    ax.axes.set_aspect("equal")
```



Do you want to see the time evolution? No problem. Let's take the velocity vector `u`, use isel with a slice selecting every 40 points in time (otherwise we would get too many figures), and plot:

[14]:
```
g = dataset.u.isel(t=slice(None, None, 40)).plot(
    x="x", y="y", col="t", row="i", cmap="turbo", rasterized=True
)

for ax in g.axes.flat:
    ax.axes.set_aspect("equal")
```



To exemplify differentiation and parallel computing capabilities, let's compute the vorticity for our dataset. We just

have one component for this 2D example, it is given by the equation:

$$\omega_z = \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}.$$

We can use xarray.DataArray.differentiate just out of the box with its second order accurate central differences. However, we can use the 4th order accurate centered scheme available at X3dDataArray.first_derivative.

We start setting the attribute boundary conditions (`BC`) for the velocity field:

[15]:
```
dataset["u"].attrs["BC"] = prm.get_boundary_condition("u")
```

and then we compute the vorticity:

[16]:
```
%%time
dataset["vort"] = (
    dataset.u.sel(i="y").x3d.first_derivative("x")
    - dataset.u.sel(i="x").x3d.first_derivative("y")
)
```

```
CPU times: user 948 ms, sys: 59.8 ms, total: 1.01 s
Wall time: 1.01 s
```

Notice the equation above computed the vorticity for the entire time series in our dataset.

We can use X3dDataArray.pencil_decomp to coarse the velocity array to a dask array, ready for parallel computing (see Using Dask with xarray). Notice that X3dDataArray.pencil_decomp applies `chunk=-1` for all coordinates listed in `args`, which means no decomposition, and `'auto'` to the others, delagating to dask the job of finding the optimal distribition. One important point here is that dask considers the dataset in this example so small that the overhead for parallel computing is not worth it. As a result, it returns with just one chunk:

[17]:
```
u_chunked = dataset.u.x3d.pencil_decomp("x", "y")
u_chunked
```

[17]:
```
<xarray.DataArray 'u' (i: 2, x: 257, y: 128, t: 201)>
dask.array<xarray-<this-array>, shape=(2, 257, 128, 201), dtype=float32, chunksize=(2,
↪ 257, 128, 201), chunktype=numpy.ndarray>
Coordinates:
  * x        (x) float64 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float64 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float64 0.0 0.75 1.5 2.25 3.0 ... 147.0 147.8 148.5 149.2 150.0
  * i        (i) object 'x' 'y'
Attributes:
    BC:       {'x': {'ncl1': 2, 'ncln': 2, 'npaire': 1}, 'y': {'ncl1': 0, 'nc...
```

Parallel computing is presented in this tutorial anyway, because X3dDataArray.pencil_decomp returns the arrays with several chunks for datasets in real scale. Each of these chunks will be computed in parallel in multi-core systems.

Just to exemplify, let's create blocks with 51 points in time, so we can use 4 cores to compute it in parallel:

[18]:
```
u_chunked = dataset.u.chunk(chunks=dict(t = 51))
u_chunked
```

[18]:
```
<xarray.DataArray 'u' (i: 2, x: 257, y: 128, t: 201)>
dask.array<xarray-<this-array>, shape=(2, 257, 128, 201), dtype=float32, chunksize=(2,
↪ 257, 128, 51), chunktype=numpy.ndarray>
Coordinates:
  * x        (x) float64 0.0 0.07812 0.1562 0.2344 ... 19.77 19.84 19.92 20.0
  * y        (y) float64 0.0 0.09375 0.1875 0.2812 ... 11.62 11.72 11.81 11.91
  * t        (t) float64 0.0 0.75 1.5 2.25 3.0 ... 147.0 147.8 148.5 149.2 150.0
```

(continues on next page)

(continued from previous page)

```
  * i        (i) object 'x' 'y'
Attributes:
    BC:       {'x': {'ncl1': 2, 'ncln': 2, 'npaire': 1}, 'y': {'ncl1': 0, 'nc...
```

Now computing the vorticity in parallel:

[19]:
```
%%time
dataset["vort"] = (
    u_chunked.sel(i="y").x3d.first_derivative("x")
    - u_chunked.sel(i="x").x3d.first_derivative("y")
).compute()
```

```
CPU times: user 3.03 s, sys: 1.34 s, total: 4.37 s
Wall time: 2.76 s
```

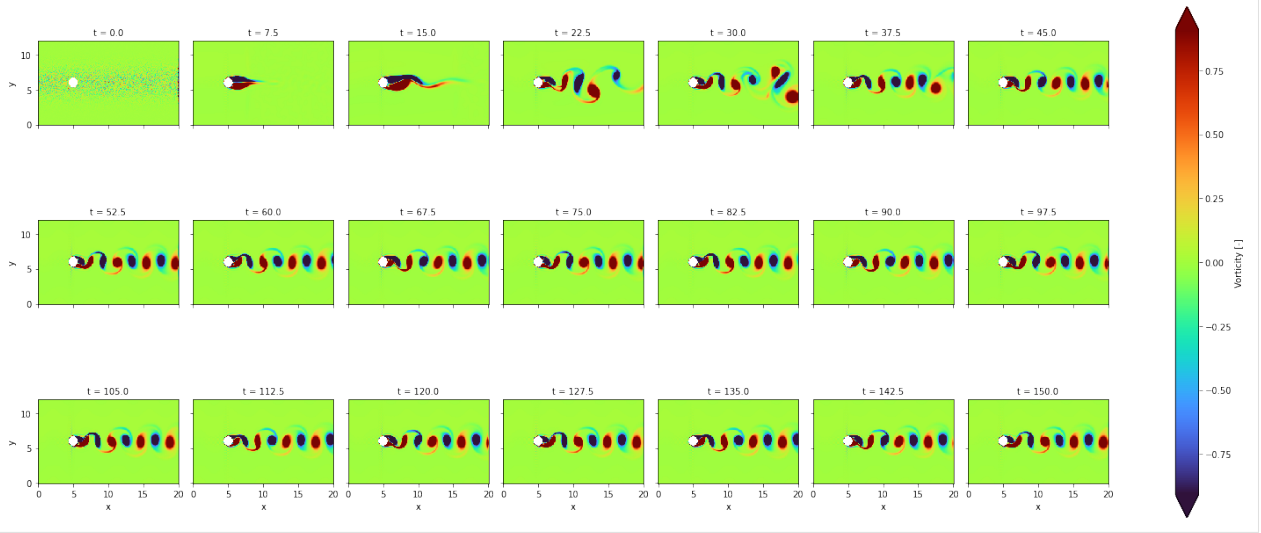
Again, remember that the dataset in this tutorial is to small that the overhead for parallel computing is not worth it. The wall time was 3 times bigger, but the code is here if you plan to try it on large scale simulations.

As usual, we can set attributes to the array we just computed:

[20]:
```
dataset["vort"].attrs = dict(name = "wz", long_name="Vorticity", units="-")
```

And plot it:

[21]:
```
g = dataset.vort.isel(t=slice(None, None, 10)).plot(
    x="x", y="y", col="t", col_wrap=7, cmap="turbo", rasterized=True, robust=True
)
for ax in g.axes.flat:
    ax.axes.set_aspect("equal")
```



Notice that xarray is built on top of Numpy, so its arrays and datasets are compatibles with many tools of the Numpy/SciPy universe. You can even access a `numpy.ndarray` object with the property `data`:

[22]:
```
dataset.epsi.data
```

[22]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
```

(continues on next page)

(continued from previous page)

```
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

You can use it for backwards compatibility with your previous post-processing tools, in this way, the transition to xcompact3d-toolbox should be easier. It is just not so effective, because we lost track of metadata like the coordinates and attributes, they are key points for data analysis with xarray.

Interactive Visualization

For an interactive experience launch this tutorial on Binder, the widgets are not responsive when disconnected from a Python application.

All the previous examples where based on matplotlib, but xarray is compatible with more options. One of them is hvPlot (see Gridded Data).

hvPlot is recommended when you are exploring your data and need a bit more interactivity.

To exemplify, let's reproduce one of the figure we did before, choosing one specific location in our mesh and looking at the time evolution of the velocity there:

[23]:
```
dataset.u.sel(x=10.0, y=6.0, method="nearest").hvplot(x="t", by="i")
```

[23]:
```
:NdOverlay   [i]
   :Curve   [t]   (u)
```

One key aspect about hvPlot is that when it gets more coordinates than it can handle in a plot, it presents the extra coordinates in widgets. So if we do not select any specific point in the domain and reproduce the same figure above, we will get widgets to select the point where we are looking:

[24]:
```
dataset.u.hvplot(x="t", by="i", widget_location="bottom")
```

[24]:
```
Column
    [0] HoloViews(DynamicMap, widget_location='bottom')
    [1] Row
        [0] HSpacer()
        [1] WidgetBox
            [0] DiscreteSlider(margin=(20, 20, 5, 20), name='y',␣
↪options=OrderedDict([('0', ...]), value=0.0, width=250)
            [1] DiscreteSlider(margin=(5, 20, 20, 20), name='x',␣
↪options=OrderedDict([('0', ...]), value=0.0, width=250)
        [2] HSpacer()
```

Here we reproduce the time evolution of the kinect energy, this time with hvPlot:

[25]:
```
dataset["kinetic_energy"].hvplot()
```

[25]:
```
:Curve   [t]   (kinetic_energy)
```

And one last example, we can see a really nice animation of the vorticity field, here in a Jupyter Notebook, with a very few lines of code:

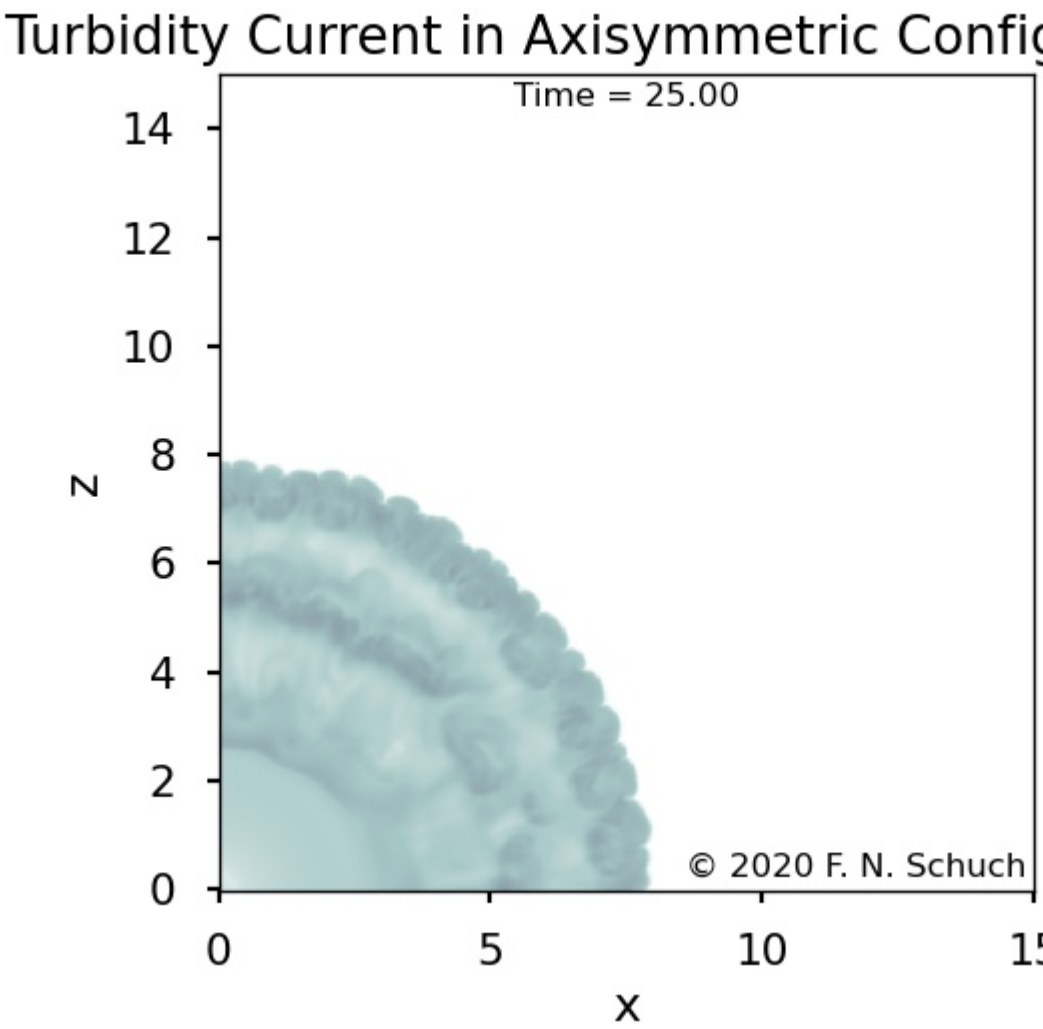
[26]:
```
dataset.sel(t = slice(40, None)).vort.hvplot(
    x="x",
    y="y",
    aspect="equal",
    clim=(-5, 5),
    rasterize=True,
    cmap="turbo",
    widget_type="scrubber",
    widget_location="bottom",
    title="Flow around a Cylinder",
    clabel=r"Vorticity [-]",
)
```

[26]:
```
Column
    [0] HoloViews(DynamicMap, widget_location='bottom', widget_type='scrubber')
    [1] Row
        [0] HSpacer()
        [1] WidgetBox
            [0] Player(end=146, width=550)
        [2] HSpacer()
```

Note: The `sel(t = slice(40, None))` in the block above is not necessary, of course, we can see the animation since the beginning. It was just used to look better at readthedocs.

4.3 Sandbox Examples

4.3.1 Turbidity Current in Axisymmetric Configuration



[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

import xcompact3d_toolbox as x3d
```

4.3.1.1 Parameters

- Numerical precision

Use `np.float64` if Xcompact3d was compiled with the flag `-DDOUBLE_PREC`, use `np.float32` otherwise.

[2]:
```
x3d.param["mytype"] = np.float32
```

- Xcompact3d's parameters

For more information about them, checkout the API reference.

[3]:
```
prm = x3d.Parameters(
    filename="input.i3d",
    # BasicParam
    itype=12,
    nx=501,
    ny=73,
    nz=501,
    xlx=20.0,
    yly=2.0,
    zlz=20.0,
    nclx1=1,
    nclxn=1,
    ncly1=2,
    nclyn=1,
    nclz1=1,
    nclzn=1,
    re=3500.0,
    init_noise=0.0125,
    dt=5e-4,
    ifirst=1,
    ilast=80000,
    numscalar=1,
    # ScalarParam
    nclxS1=1,
    nclxSn=1,
    nclyS1=1,
    nclySn=1,
    nclzS1=1,
    nclzSn=1,
    sc=[1.0],
    ri=[0.5],
    uset=[0.0],
    cp=[1.0],
)
```

4.3.1.2 Setup

Everything needed is in one Dataset (see API reference):

[4]:
```
ds = x3d.init_dataset(prm)
```

Let's see it, data and attributes are attached, try to interact with the icons:

[5]:
```
ds
```

[5]:
```
<xarray.Dataset>
Dimensions:  (x: 501, y: 73, z: 501, n: 1)
Coordinates:
  * x        (x) float32 0.0 0.04 0.08 0.12 0.16 ... 19.88 19.92 19.96 20.0
  * y        (y) float32 0.0 0.02778 0.05556 0.08333 ... 1.917 1.944 1.972 2.0
  * z        (z) float32 0.0 0.04 0.08 0.12 0.16 ... 19.88 19.92 19.96 20.0
  * n        (n) int32 1
Data variables:
```

(continues on next page)

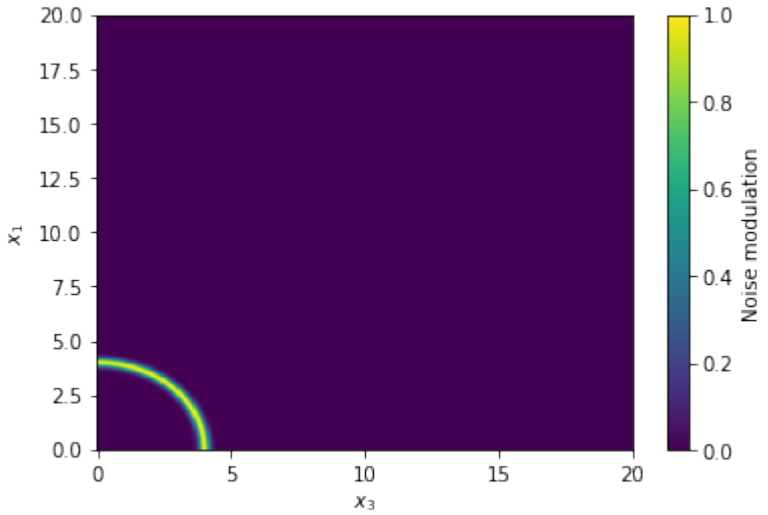
(continued from previous page)

```
    ux       (x, y, z) float32 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    uy       (x, y, z) float32 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    uz       (x, y, z) float32 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    phi      (n, x, y, z) float32 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
```

Initial Condition

A random noise will be applied to the initial velocity field, we start creating a modulation function `mod`, to apply it just near the fresh/turbidity interface:

[6]:
```
# Position of the initial interface in the polar coordinate
r0 = 4.0

# Random noise with fixed seed,
# important for reproducibility, development and debugging
if prm.iin == 2:
    np.random.seed(seed=67)

radius = np.sqrt(ds.x ** 2 + ds.z ** 2.0)

mod = np.exp(-25.0 * (radius - r0) ** 2.0)

# This attribute will be shown at the colorbar
mod.attrs["long_name"] = "Noise modulation"

mod.plot();
```



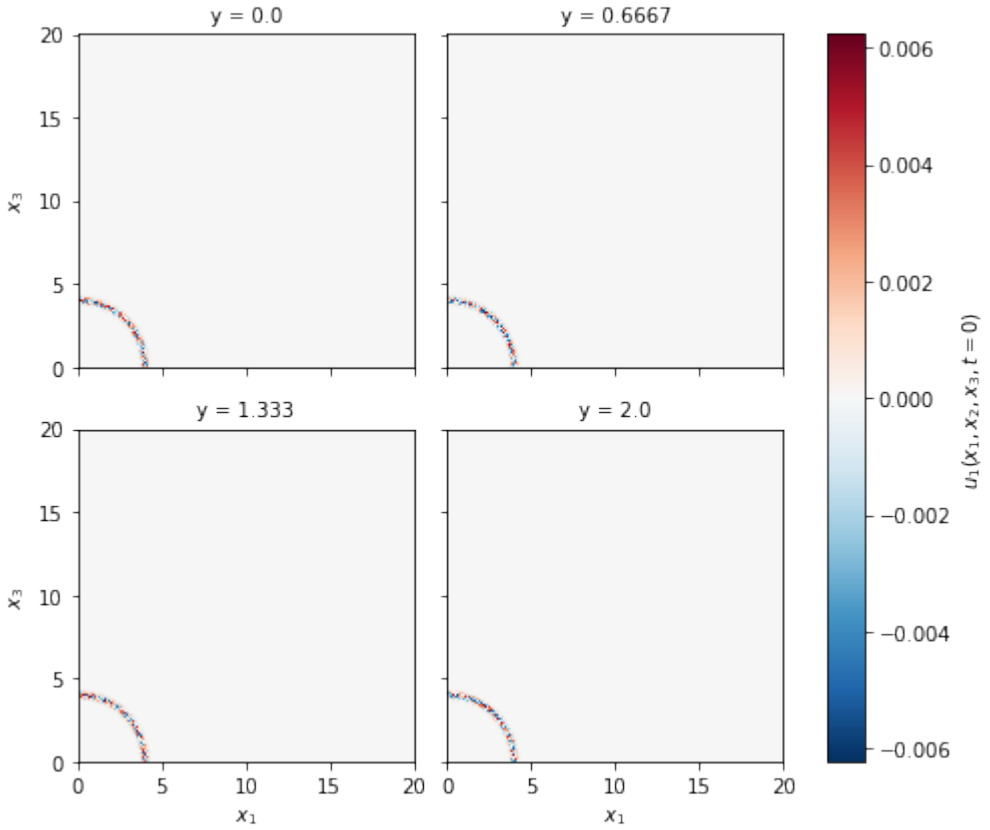
Now we reset velocity fields `ds[key] *= 0.0`, just to guarantee consistency in the case of multiple executions of this cell. Notice that `ds[key] = 0.0` may overwrite all the metadata contained in the array, so it should be avoided.

We then add a random number array with the right shape and multiply by the noise amplitude at the initial condition `init_noise` and multiply again by our modulation function `mod`, defined previously.

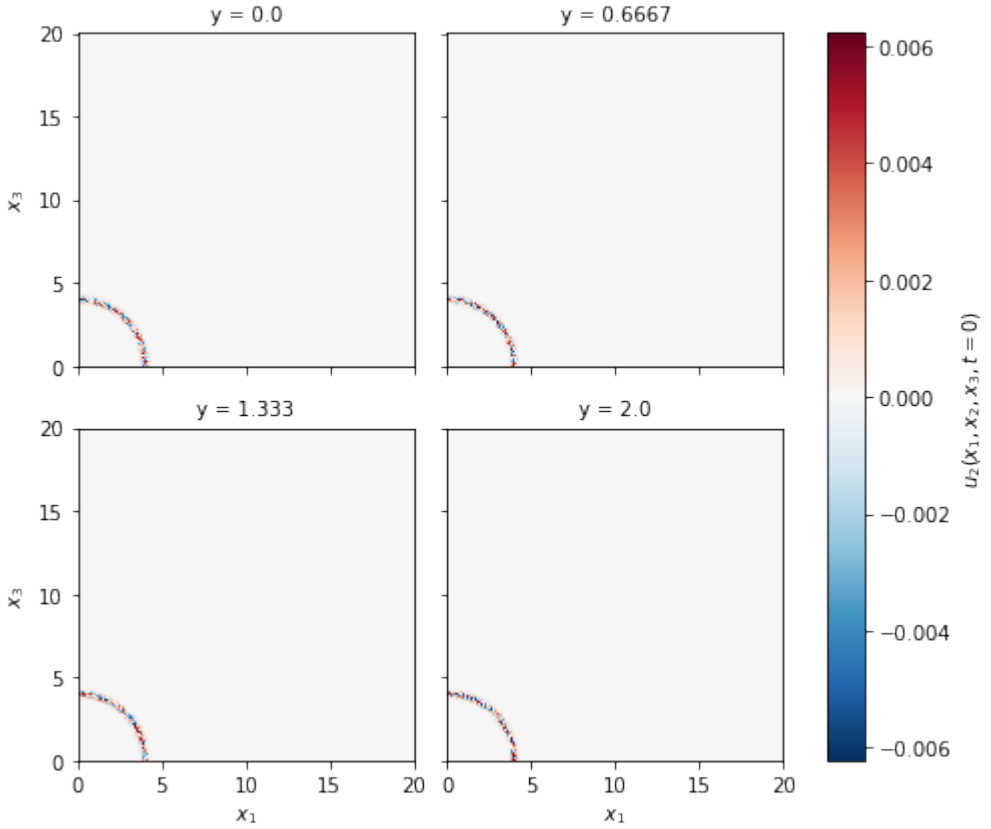
Plotting a `xarray.DataArray` is as simple as `da.plot()` (see its user guide), I'm adding extra options just to exemplify how easily we can slice the vertical coordinate and produce multiple plots:

[7]:
```
for key in "ux uy uz".split():
    #
    print(ds[key].attrs["name"])
    #
    ds[key] *= 0.0
    #
    ds[key] += prm.init_noise * ((np.random.random(ds[key].shape) - 0.5))
    ds[key] *= mod
    #
    ds[key].sel(y=slice(None, None, ds.y.size // 3)).plot(
        x="x", y="z", col="y", col_wrap=2
    )
    plt.show()

plt.close("all")
```

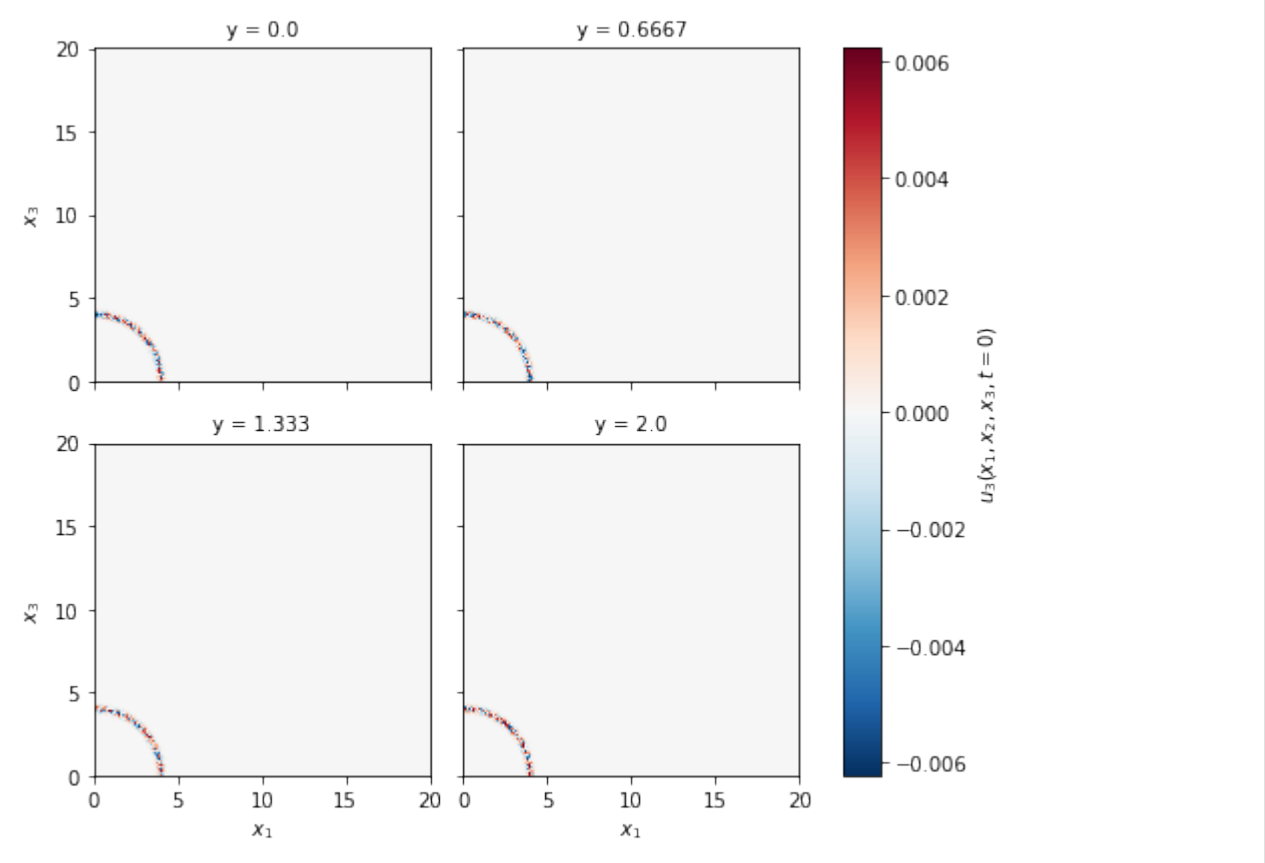
```
Initial Condition for Streamwise Velocity
```




```
Initial Condition for Vertical Velocity
```



Initial Condition for Spanwise Velocity



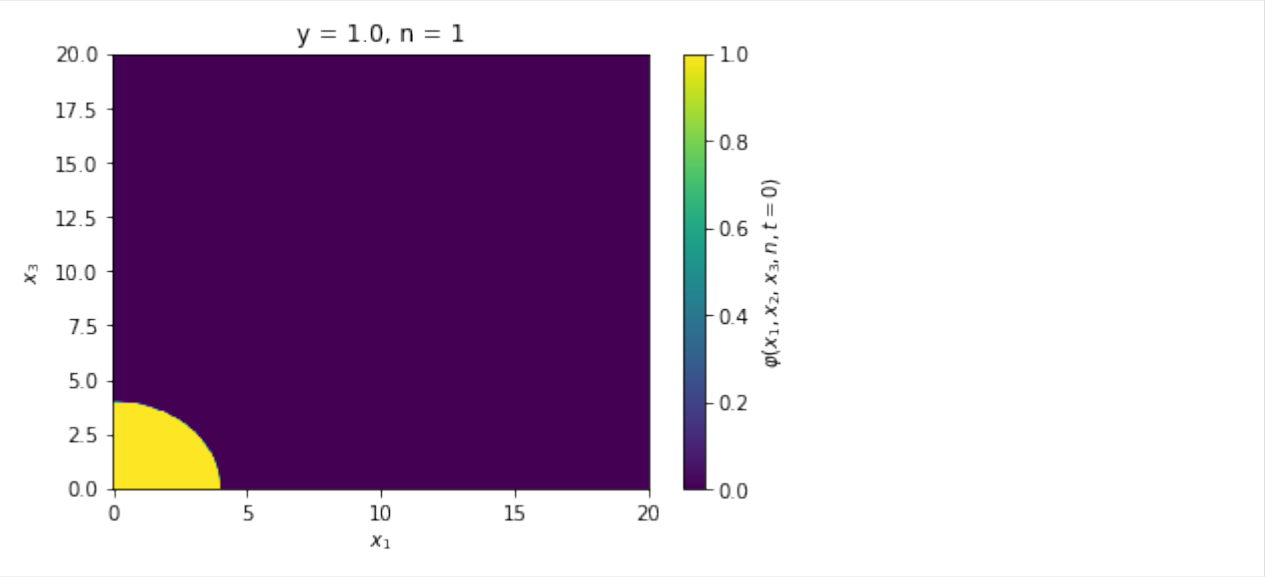
A smooth transition at the interface fresh/turbidity fluid is used for the initial concentration(s) field(s), it is defined in the polar coordinates as:

$$\varphi_n = c_{0,n}\frac{1}{2}\left(1 - \tanh\left((r - r_0)\sqrt{Sc_n Re}\right)\right).$$

The code block includes the same procedures, we reset the scalar field `ds['phi'] *= 0.0`, just to guarantee consistency, we compute the equation above, we add it to the array and make a plot:

[8]:
```
# Concentration

print(ds["phi"].attrs["name"])

ds["phi"] *= 0.0

for n in range(prm.numscalar):
    #
    fun = 0.5 * prm.cp[n] * (1.0 - np.tanh((radius - r0) * (prm.sc[n] * prm.re) ** 0.
↪5))
    #
    ds["phi"][dict(n=n)] += fun
    #
    ds.phi.isel(n=n).sel(y=prm.yly / 2.0).T.plot()
    plt.show()

plt.close("all")
```

```
Initial Condition for Scalar field(s)
```



4.3.1.3 Writing to disc

is as simple as:

[9]:
```
prm.dataset.write(ds)
```

[10]:
```
prm.write()
```

4.3.1.4 Running the Simulation

It was just to show the capabilities of `xcompact3d_toolbox.sandbox`, keep in mind the aspects of numerical stability of our Navier-Stokes solver. **It is up to the user to find the right set of numerical and physical parameters**.

Make sure that the compiling flags and options at `Makefile` are what you expect. Then, compile the main code at the root folder with `make`.

And finally, we are good to go:

```
mpirun -n [number of cores] ./xcompact3d |tee log.out
```

4.3.2 Flow Around a Complex Body

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

import xcompact3d_toolbox as x3d
```

4.3.2.1 Parameters

- Numerical precision

Use `np.float64` if Xcompact3d was compiled with the flag `-DDOUBLE_PREC`, use `np.float32` otherwise.

[2]:
```
x3d.param["mytype"] = np.float64
```

- Xcompact3d's parameters

For more information about them, checkout the API reference.

[3]:
```
prm = x3d.Parameters(
    filename="input.i3d",
    itype=12,
    p_row=0,
    p_col=0,
    nx=257,
    ny=129,
    nz=32,
    xlx=15.0,
    yly=10.0,
    zlz=3.0,
    nclx1=2,
    nclxn=2,
    ncly1=1,
    nclyn=1,
    nclz1=0,
    nclzn=0,
    iin=1,
    re=300.0,
    init_noise=0.0125,
    inflow_noise=0.0125,
    dt=0.0025,
    ifirst=1,
    ilast=45000,
    ilesmod=1,
    iibm=2,
    nu0nu=4.0,
    cnu=0.44,
    irestart=0,
    icheckpoint=45000,
    ioutput=200,
    iprocessing=50,
    jles=4,
)
```

4.3.2.2 Setup

Geometry

Everything needed is in one dictionary of Arrays (see API reference):

[4]:
```
epsi = x3d.init_epsi(prm)
```

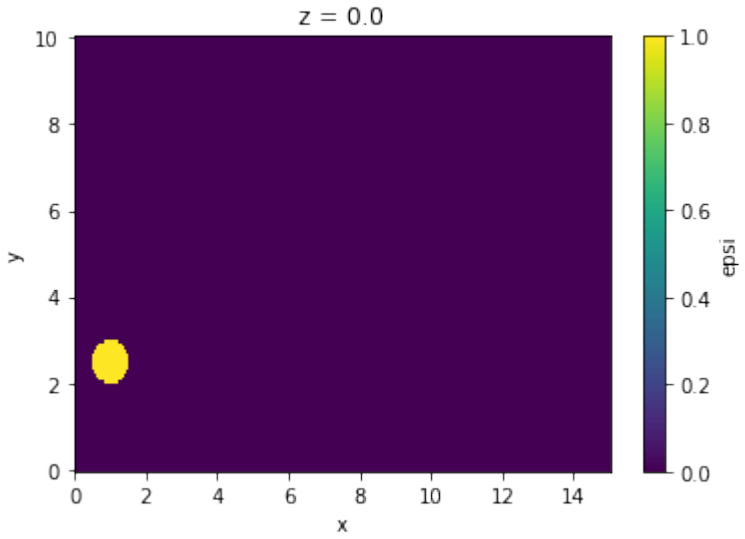
The four ϵ matrices are stored in a dictionary:

[5]:
```
epsi.keys()
```

[5]:
```
dict_keys(['epsi', 'xepsi', 'yepsi', 'zepsi'])
```

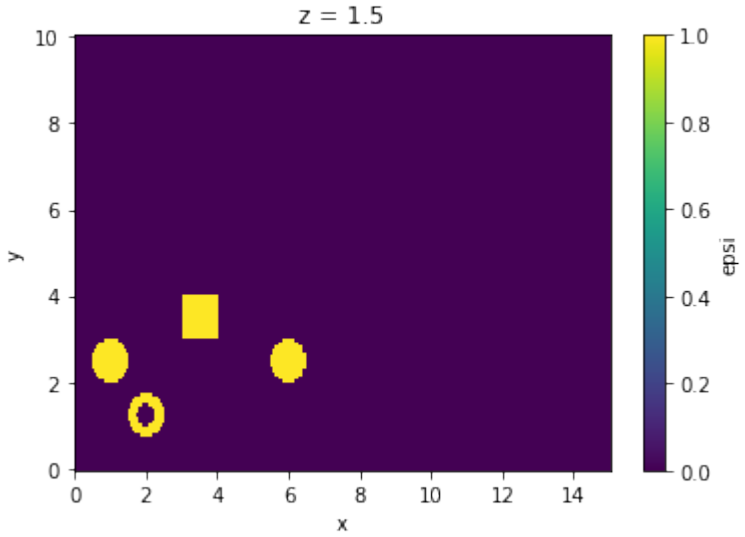
Just to exemplify, we can draw and plot a cylinder. Make sure to apply the same operation over all arrays in the dictionary. Plotting a `xarray.DataArray` is as simple as `da.plot()` (see its user guide), I'm adding extra options just to exemplify how easily we can select one value in z and make a 2D plot:

[6]:
```
for key in epsi.keys():
    epsi[key] = epsi[key].geo.cylinder(x=1, y=prm.yly / 4.0)

epsi["epsi"].sel(z=0, method="nearest").plot(x="x");
```



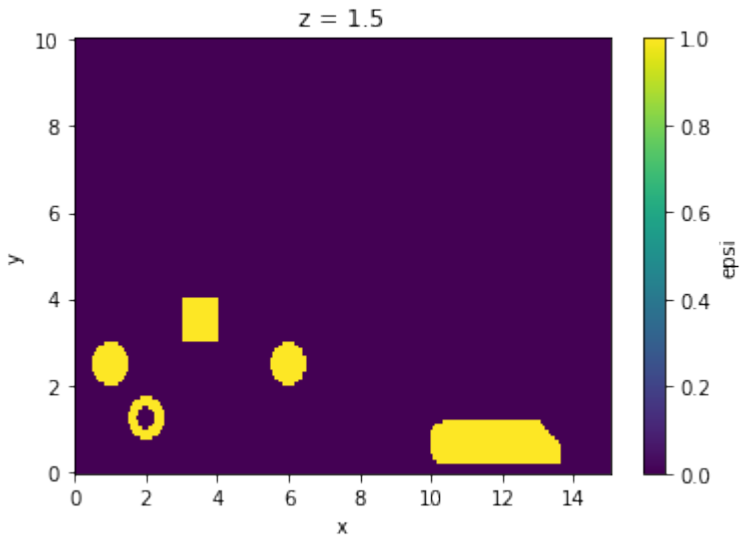
Notice that the geometries are added by default, however, we can revert it by setting `remp=False`. We can execute several methods in a chain, resulting in more complex geometries.

[7]:
```
for key in epsi.keys():
    epsi[key] = (
        epsi[key]
        .geo.cylinder(x=2, y=prm.yly / 8.0)
        .geo.cylinder(x=2, y=prm.yly / 8.0, radius=0.25, remp=False)
        .geo.sphere(x=6, y=prm.yly / 4, z=prm.zlz / 2.0)
        .geo.box(x=[3, 4], y=[3, 4])
    )

epsi["epsi"].sel(z=prm.zlz / 2, method="nearest").plot(x="x");
```



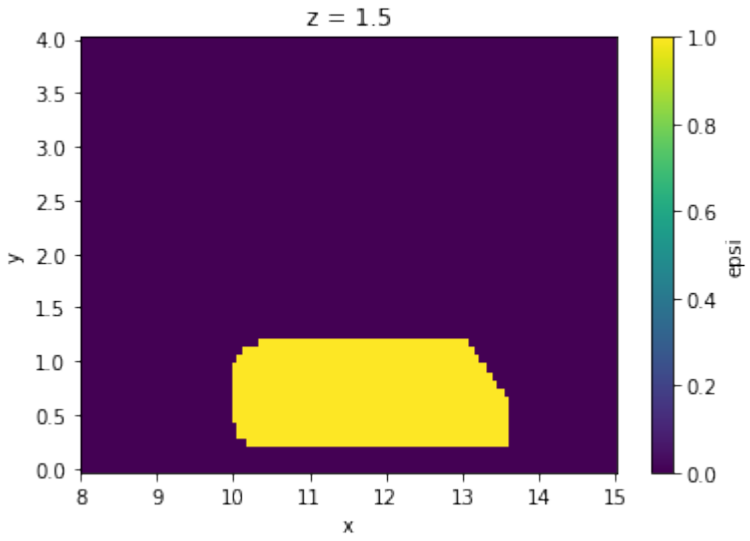
Other example, Ahmed body:

[8]:
```
for key in epsi.keys():
    epsi[key] = epsi[key].geo.ahmed_body(x=10, wheels=False)

epsi["epsi"].sel(z=prm.zlz / 2, method="nearest").plot(x="x");
```



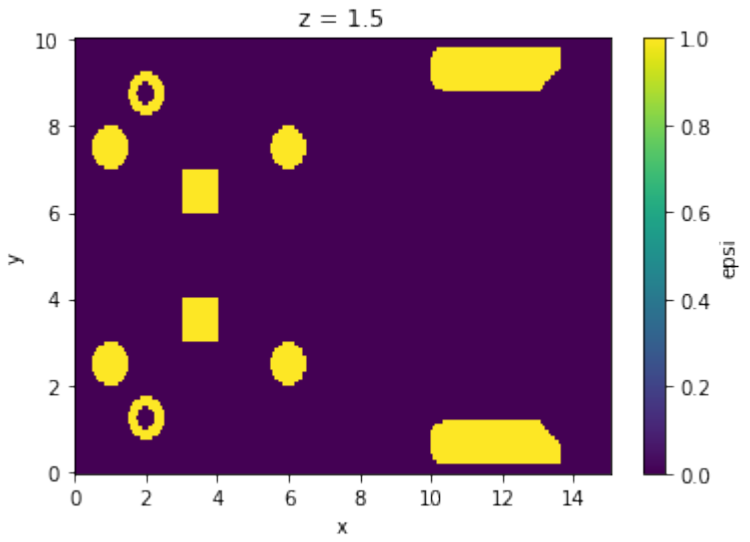
Zooming in:

[9]:
```
epsi["epsi"].sel(x=slice(8, None), y=slice(None, 4)).sel(
    z=prm.zlz / 2, method="nearest"
).plot(x="x");
```



And just as an example, we can mirror it:

[10]:
```
for key in epsi.keys():
    epsi[key] = epsi[key].geo.mirror("y")

epsi["epsi"].sel(z=prm.zlz / 2, method="nearest").plot(x="x");
```



It was just to show the capabilities of `xcompact3d_toolbox.sandbox`, you can use it to build many different geometries and arrange them in many ways. However, keep in mind the aspects of numerical stability of our Navier-Stokes solver, **it is up to the user to find the right set of numerical and physical parameters**.

For a complete description about the available geometries see Api reference. Notice that you combine them for the creation of unique geometries, or even create your own routines for your own objects.

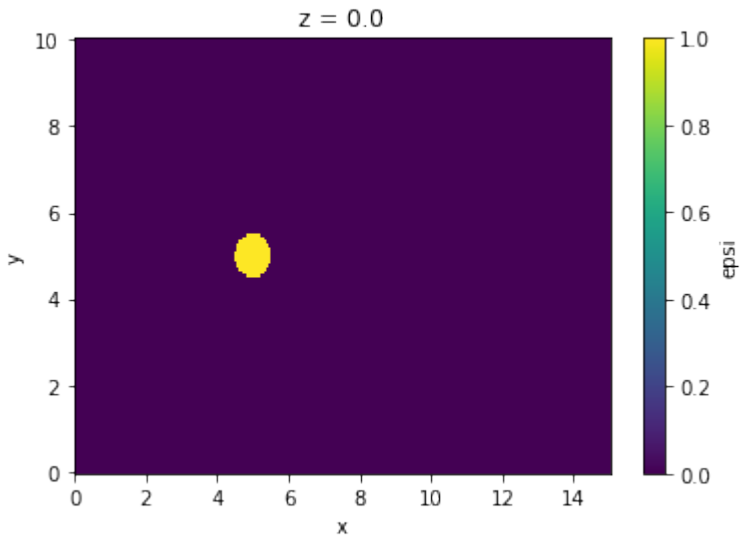
So, let's start over with a simpler geometry:

[11]:
```
epsi = x3d.sandbox.init_epsi(prm)
```

(continues on next page)

(continued from previous page)

```
for key in epsi.keys():
    epsi[key] = epsi[key].geo.cylinder(x=prm.xlx / 3, y=prm.yly / 2)

epsi["epsi"].sel(z=0, method="nearest").plot(x="x")
plt.show();
```



The next step is to produce all the auxiliary files describing the geometry, so then Xcompact3d can read them:

[12]:
```
%%time
dataset = x3d.gene_epsi_3D(epsi, prm)

prm.nobjmax = dataset.obj.size

dataset
```

```
x
       nobjraf : 1
    nobjmaxraf : 1
           bug : 0

y
       nobjraf : 1
    nobjmaxraf : 1
           bug : 0

z
       nobjraf : 1
    nobjmaxraf : 1
           bug : 0

number of points with potential problem in x : 0
number of points with potential problem in y : 0
number of points with potential problem in z : 0

Writing...
Wall time: 6.01 s
```

[12]:
```
<xarray.Dataset>
Dimensions:       (obj_aux: 2, obj: 1, x: 257, y: 129, z: 32)
Coordinates:
  * obj_aux       (obj_aux) int32 -1 0
  * obj           (obj) int32 0
  * x             (x) float64 0.0 0.05859 0.1172 0.1758 ... 14.88 14.94 15.0
  * y             (y) float64 0.0 0.07812 0.1562 0.2344 ... 9.844 9.922 10.0
  * z             (z) float64 0.0 0.09375 0.1875 0.2812 ... 2.719 2.812 2.906
Data variables: (12/28)
    epsi          (x, y, z) bool False False False False ... False False False
    nobj_x        (y, z) int64 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0
    nobjmax_x     int64 1
    nobjraf_x     (y, z) int64 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0
    nobjmaxraf_x  int64 1
    ibug_x        int64 0
    ...            ...
    nxipif_y      (x, z, obj_aux) int64 2 2 2 2 2 2 2 2 ... 2 2 2 2 2 2 2 2 2
    nxfpif_y      (x, z, obj_aux) int64 128 2 128 2 128 2 ... 128 2 128 2 128 2
    xi_z          (x, y, obj) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    xf_z          (x, y, obj) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    nxipif_z      (x, y, obj_aux) int64 2 2 2 2 2 2 2 2 2 ... 2 2 2 2 2 2 2 2 2
    nxfpif_z      (x, y, obj_aux) int64 31 2 31 2 31 2 31 2 ... 2 31 2 31 2 31 2
```

Boundary Condition

Everything needed is in one Dataset (see API reference):

[13]:
```
ds = x3d.init_dataset(prm)
```

Let's see it, data and attributes are attached, try to interact with the icons:

[14]:
```
ds
```

[14]:
```
<xarray.Dataset>
Dimensions:       (x: 257, y: 129, z: 32, n: 0)
Coordinates:
  * x             (x) float64 0.0 0.05859 0.1172 0.1758 ... 14.88 14.94 15.0
  * y             (y) float64 0.0 0.07812 0.1562 0.2344 ... 9.844 9.922 10.0
  * z             (z) float64 0.0 0.09375 0.1875 0.2812 ... 2.719 2.812 2.906
  * n             (n) float64
Data variables:
    bxx1          (y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    bxy1          (y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    bxz1          (y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    noise_mod_x1  (y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    ux            (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    uy            (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    uz            (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
```

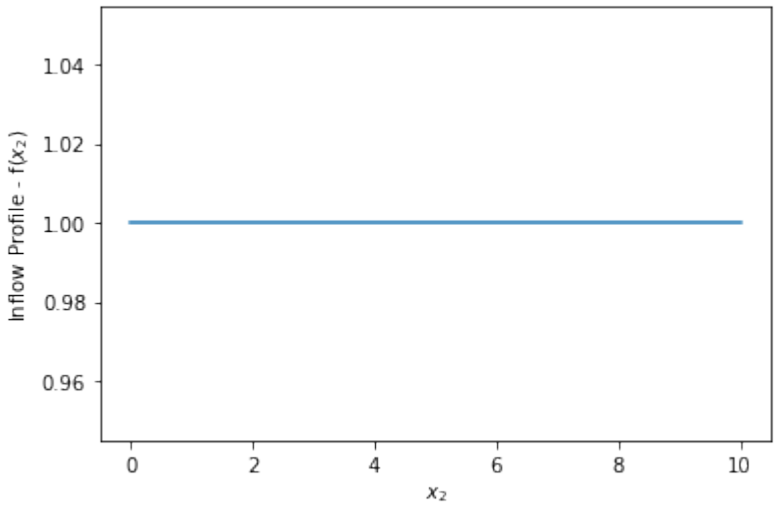
Inflow profile: Since the boundary conditions for velocity at the top and at the bottom are free-slip in this case (`ncly1=nclyn=1`), the inflow profile for streamwise velocity is just 1 everywhere:

[15]:
```
fun = xr.ones_like(ds.y)

# This attribute will be shown in the figure
fun.attrs["long_name"] = r"Inflow Profile - f($x_2$)"
```

(continues on next page)

(continued from previous page)

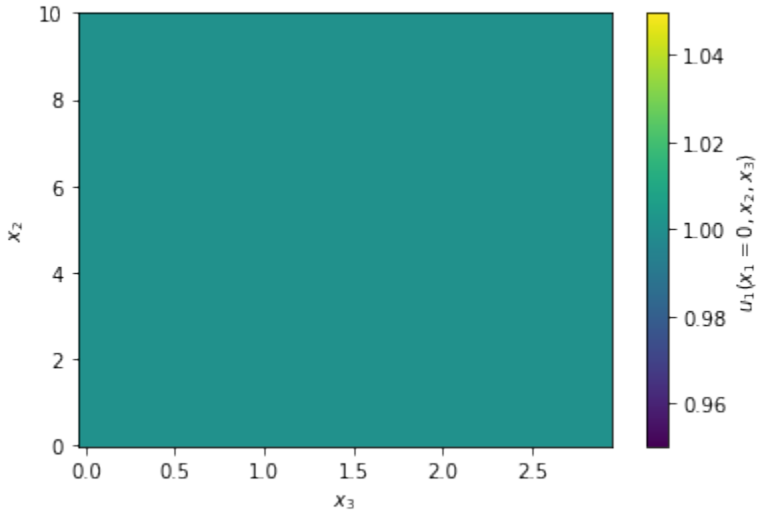
```
fun.plot();
```



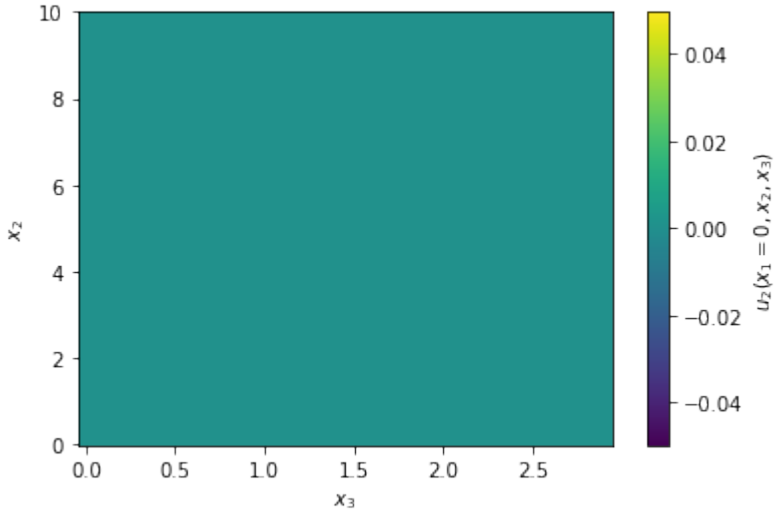
Now, we reset the inflow planes `ds[key] *= 0.0`, just to guarantee consistency in case of multiple executions of this cell. Notice that `ds[key] = 0.0` may overwrite all the metadata contained in the array, so it should be avoided. Then, we add the inflow profile to the streamwise componente and plot them for reference:

[16]:
```
for key in "bxx1 bxy1 bxz1".split():
    #
    print(ds[key].attrs["name"])
    #
    ds[key] *= 0.0
    #
    if key == "bxx1":
        ds[key] += fun
    #
    ds[key].plot()
    plt.show()

plt.close("all")
```

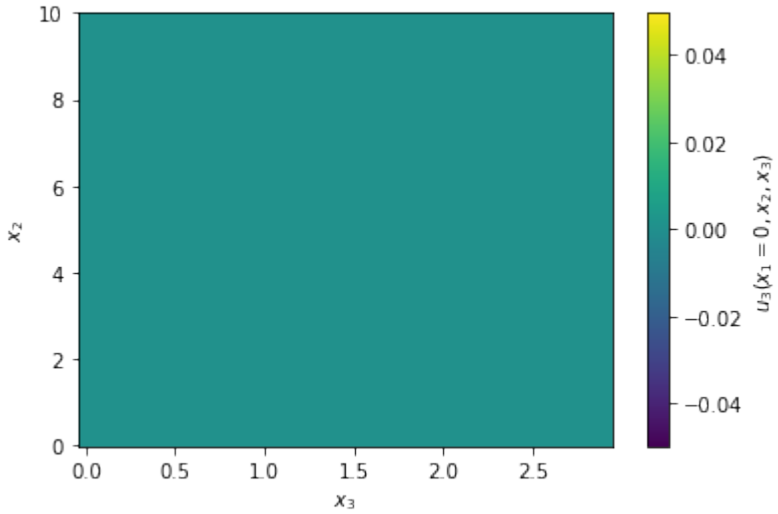
```
Inflow Plane for Streamwise Velocity
```



Inflow Plane for Vertical Velocity



Inflow Plane for Spanwise Velocity

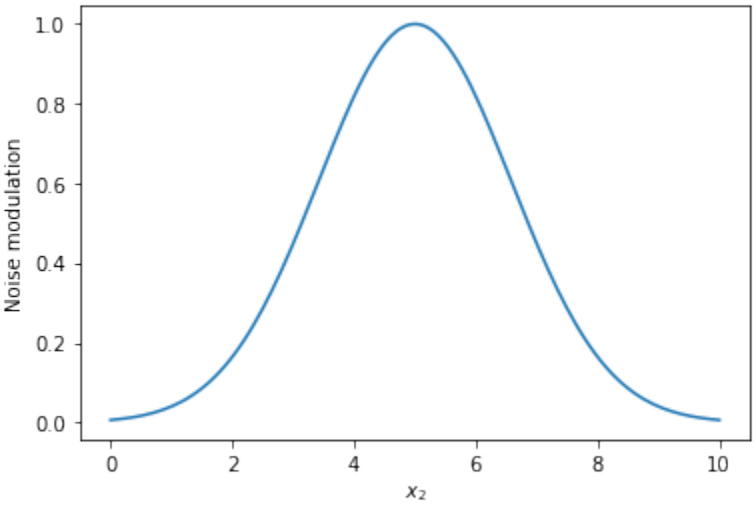


A random noise will be applied at the inflow boundary, we can create a modulation function `mod` to control were it will be applied. In this case, we will concentrate the noise near the center region and make it zero were $y = 0$ and $y = L_y$. The domain is periodic in z `nclz1=nclzn=0`, so there is no need to make `mod` functions of z. The functions looks like:

$$\text{mod} = \exp\left(-0.2(y - 0.5L_y)^2\right).$$

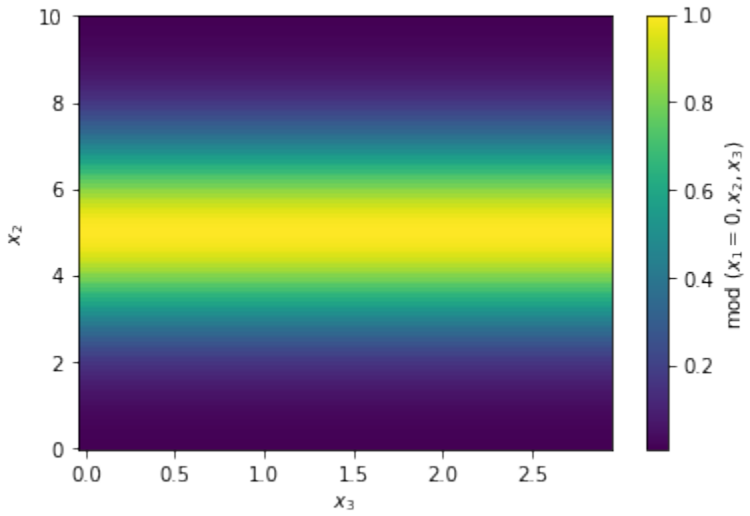
See the code:

[17]:
```
# Random noise with fixed seed,
# important for reproducibility, development and debugging
if prm.iin == 2:
    np.random.seed(seed=67)

mod = np.exp(-0.2 * (ds.y - ds.y[-1] * 0.5) ** 2.0)

# This attribute will be shown in the figure
mod.attrs["long_name"] = "Noise modulation"

mod.plot();
```



Again, we reset the array `ds['noise_mod_x1'] *= 0.0`, just to guarantee consistency in case of multiple executions of this cell. Notice that `ds['noise_mod_x1'] *= 0.0` may overwrite all the metadata contained in the array, so it should be avoided. Then, we add the modulation profile to the proper array and plot it for reference:

[18]:
```
ds["noise_mod_x1"] *= 0.0
ds["noise_mod_x1"] += mod
ds.noise_mod_x1.plot();
```



Notice one of the many advantages of using xarray, `mod`, with shape (`ny`), was automatically broadcasted for every point in `z` into `ds.noise_mod_x1`, with shape (`ny`, `nz`).

Initial Condition

Now we reset velocity fields `ds[key] *= 0.0`, just to guarantee consistency in the case of multiple executions of this cell.

We then add a random number array with the right shape, multiply by the noise amplitude at the initial condition `init_noise` and multiply again by our modulation function `mod`, defined previously. Finally, we add the streamwise profile `fun` to `ux` and make the plots for reference, I'm adding extra options just to exemplify how easily we can slice the spanwise coordinate and produce multiple plots:

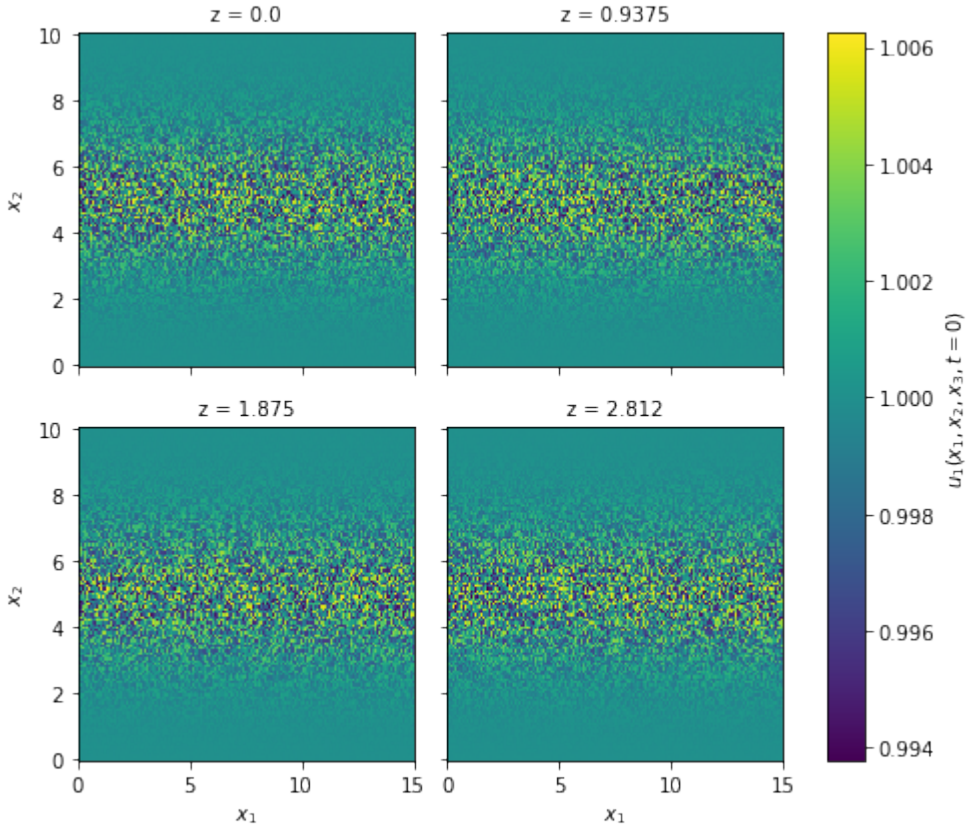
[19]:
```
for key in "ux uy uz".split():
    #
    print(ds[key].attrs["name"])
    #
    ds[key] *= 0.0
    ds[key] += prm.init_noise * ((np.random.random(ds[key].shape) - 0.5))
    ds[key] *= mod
    #
    if key == "ux":
        ds[key] += fun
    #
    ds[key].sel(z=slice(None, None, ds.z.size // 3)).plot(
        x="x", y="y", col="z", col_wrap=2
    )
    plt.show()
    #
```

(continues on next page)

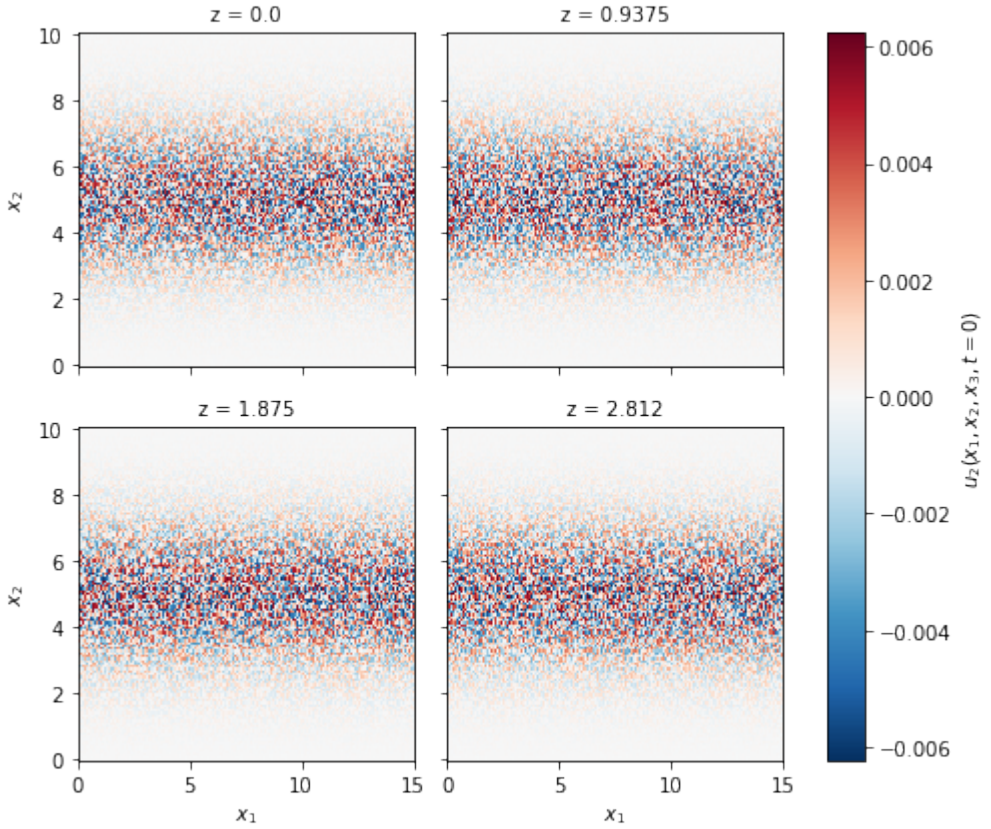
(continued from previous page)

```
plt.close("all")
```

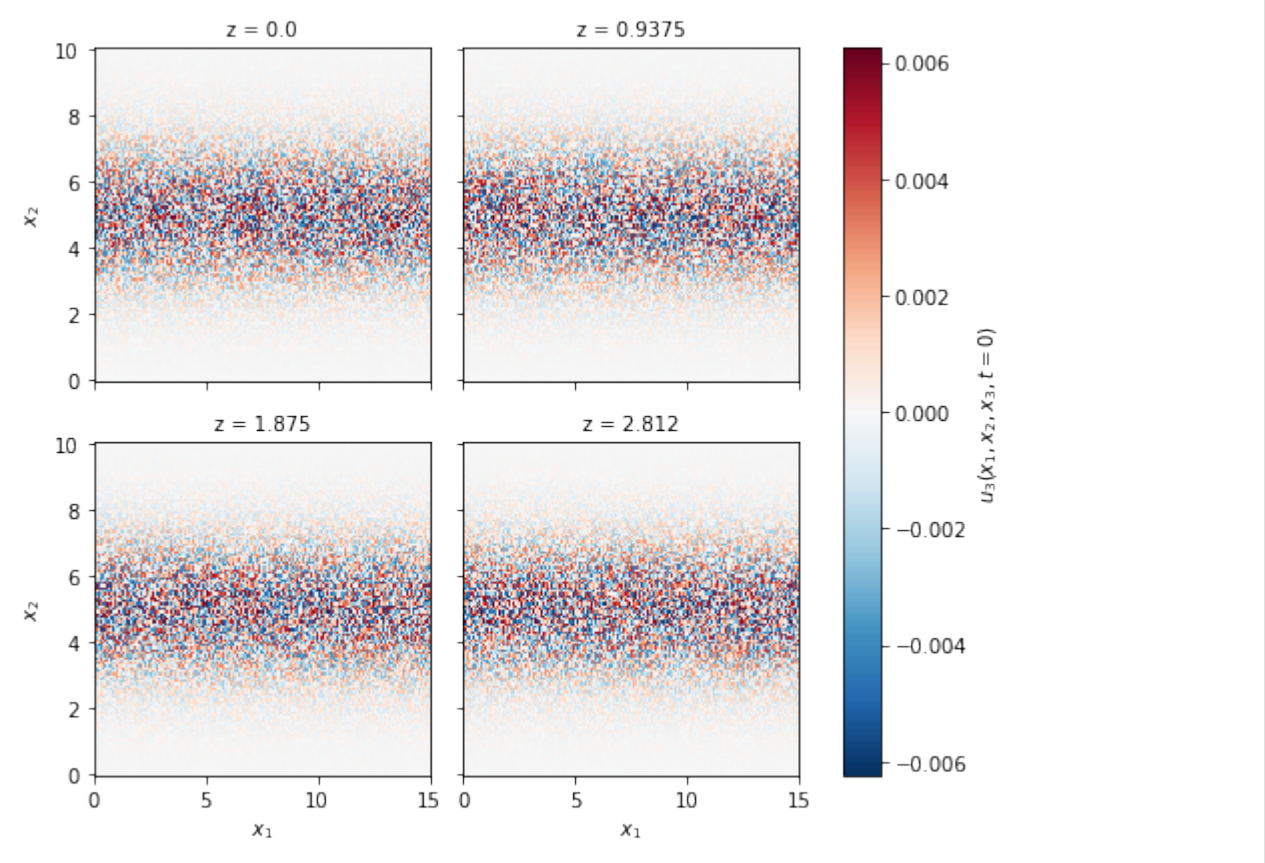
```
Initial Condition for Streamwise Velocity
```




```
Initial Condition for Vertical Velocity
```



Initial Condition for Spanwise Velocity



4.3.2.3 Writing to disc

is as simple as:

[20]:
```
prm.dataset.write(ds)
```

[21]:
```
prm.write()
```

4.3.2.4 Running the Simulation

It was just to show the capabilities of `xcompact3d_toolbox.sandbox`, keep in mind the aspects of numerical stability of our Navier-Stokes solver. **It is up to the user to find the right set of numerical and physical parameters**.

Make sure that the compiling flags and options at `Makefile` are what you expect. Then, compile the main code at the root folder with `make`.

And finally, we are good to go:

```
mpirun -n [number of cores] ./xcompact3d |tee log.out
```

4.3.3 Flow Around a Square and Flow Visualization with Passive Scalar

The no-flux boundary condition for the scalar field(s) at the slid/fluid interface is experimental, and it is still not available at XCompact3d's main repository.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

import xcompact3d_toolbox as x3d
```

4.3.3.1 Parameters

- Numerical precision

Use `np.float64` if Xcompact3d was compiled with the flag `-DDOUBLE_PREC`, use `np.float32` otherwise.

[2]:
```
x3d.param["mytype"] = np.float64
```

- Xcompact3d's parameters

For more information about them, checkout the API reference.

[3]:
```
prm = x3d.Parameters(
    filename="input.i3d",
    # BasicParam
    itype=12,
    p_row=0,
    p_col=0,
    nx=257,
    ny=129,
    nz=32,
    xlx=15.0,
    yly=10.0,
    zlz=3.0,
    nclx1=2,
    nclxn=2,
    ncly1=1,
    nclyn=1,
    nclz1=0,
    nclzn=0,
    iin=1,
    re=300.0,
    init_noise=0.0125,
    inflow_noise=0.0125,
    dt=0.0025,
    ifirst=1,
    ilast=90000,
    ilesmod=1,
    iibm=2, # This is experimental, not available at the main repo
    # NumOptions
    nu0nu=4.0,
    cnu=0.44,
    # InOutParam
    irestart=0,
    icheckpoint=45000,
    ioutput=500,
```

(continues on next page)

(continued from previous page)

```
    iprocessing=100,
    # LESModel
    jles=4,
    # ScalarParam
    numscalar=1,
    nclxS1=2,
    nclxSn=2,
    nclyS1=1,
    nclySn=1,
    nclzS1=0,
    nclzSn=0,
    sc=[1.0],
    ri=[0.0],  # Zero for numerical dye
    uset=[0.0],  # Zero for numerical dye
    cp=[1.0],
    #iibmS=3, # This is experimental, not available at the main repo
)
```

4.3.3.2 Setup

Geometry

Everything needed is in one dictionary of Arrays (see API reference):

[4]:
```
epsi = x3d.init_epsi(prm)
```

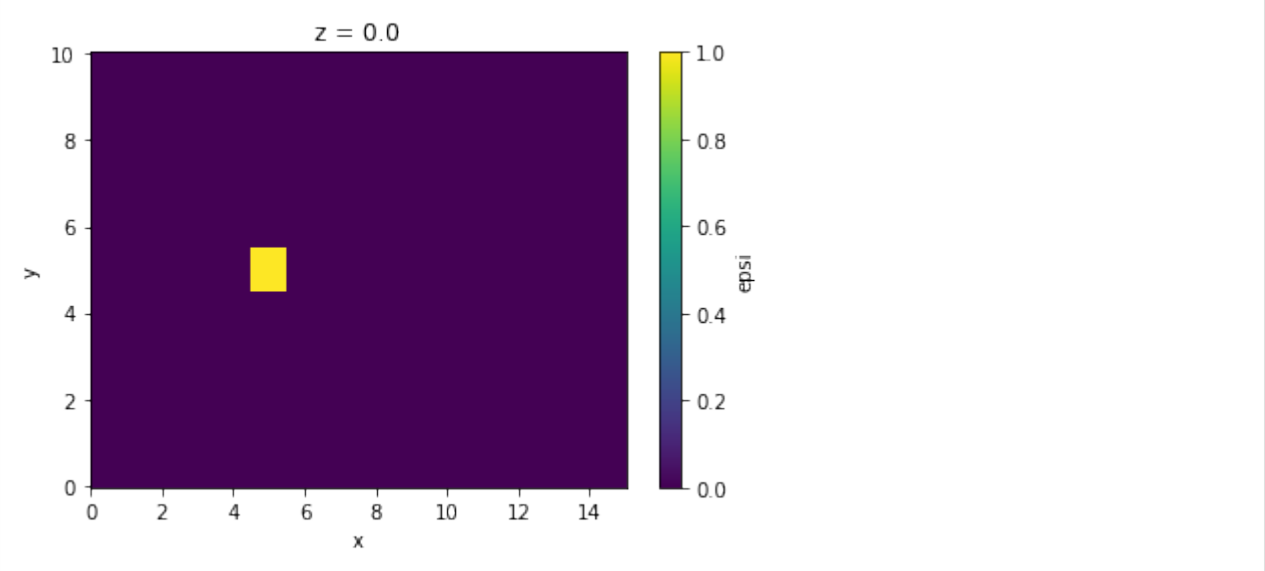
The four ϵ matrices are stored in a dictionary:

[5]:
```
epsi.keys()
```

[5]:
```
dict_keys(['epsi', 'xepsi', 'yepsi', 'zepsi'])
```

Now we draw a square:

[6]:
```
# Here we set the center
center = dict(x=prm.xlx / 3.0, y=prm.yly / 2.0)

# And apply geo.box over the four arrays
for key in epsi.keys():
    epsi[key] = epsi[key].geo.box(
        x=[center["x"] - 0.5, center["x"] + 0.5],
        y=[center["y"] - 0.5, center["y"] + 0.5],
    )

# A quickie plot for reference
epsi["epsi"].sel(z=0, method="nearest").plot(x="x");
```



Curved surfaces are not supported (yet?) for the no-flux condition for the scalar field(s) at the solid/fluid interface, so let's stay with the square.

The next step is to produce all the auxiliary files describing the geometry, so then Xcompact3d can read them:

[7]:
```
%%time
dataset = x3d.gene_epsi_3D(epsi, prm)

prm.nobjmax = dataset.obj.size

dataset
```

```
x
       nobjraf : 1
    nobjmaxraf : 1
           bug : 0

y
       nobjraf : 1
    nobjmaxraf : 1
           bug : 0

z
       nobjraf : 1
    nobjmaxraf : 1
           bug : 0

number of points with potential problem in x : 0
number of points with potential problem in y : 0
number of points with potential problem in z : 0

Writing...
Wall time: 5.32 s
```

[7]:
```
<xarray.Dataset>
Dimensions:       (obj_aux: 2, obj: 1, x: 257, y: 129, z: 32)
```

(continues on next page)

(continued from previous page)

```
Coordinates:
  * obj_aux       (obj_aux) int32 -1 0
  * obj           (obj) int32 0
  * x             (x) float64 0.0 0.05859 0.1172 0.1758 ... 14.88 14.94 15.0
  * y             (y) float64 0.0 0.07812 0.1562 0.2344 ... 9.844 9.922 10.0
  * z             (z) float64 0.0 0.09375 0.1875 0.2812 ... 2.719 2.812 2.906
Data variables: (12/28)
    epsi          (x, y, z) bool False False False False ... False False False
    nobj_x        (y, z) int64 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0
    nobjmax_x     int64 1
    nobjraf_x     (y, z) int64 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0
    nobjmaxraf_x  int64 1
    ibug_x        int64 0
    ...            ...
    nxipif_y      (x, z, obj_aux) int64 2 2 2 2 2 2 2 2 2 ... 2 2 2 2 2 2 2 2 2
    nxfpif_y      (x, z, obj_aux) int64 128 2 128 2 128 2 ... 128 2 128 2 128 2
    xi_z          (x, y, obj) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    xf_z          (x, y, obj) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    nxipif_z      (x, y, obj_aux) int64 2 2 2 2 2 2 2 2 2 ... 2 2 2 2 2 2 2 2 2
    nxfpif_z      (x, y, obj_aux) int64 31 2 31 2 31 2 31 2 ... 2 31 2 31 2 31 2
```

Boundary Condition

Everything needed is in one Dataset (see API reference):

[8]:
```
ds = x3d.init_dataset(prm)
```

Let's see it, data and attributes are attached, try to interact with the icons:

[9]:
```
ds
```

[9]:
```
<xarray.Dataset>
Dimensions:       (x: 257, y: 129, z: 32, n: 1)
Coordinates:
  * x             (x) float64 0.0 0.05859 0.1172 0.1758 ... 14.88 14.94 15.0
  * y             (y) float64 0.0 0.07812 0.1562 0.2344 ... 9.844 9.922 10.0
  * z             (z) float64 0.0 0.09375 0.1875 0.2812 ... 2.719 2.812 2.906
  * n             (n) int32 1
Data variables:
    bxx1          (y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    bxy1          (y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    bxz1          (y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    noise_mod_x1  (y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    bxphi1        (n, y, z) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    ux            (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    uy            (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    uz            (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    phi           (n, x, y, z) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
```

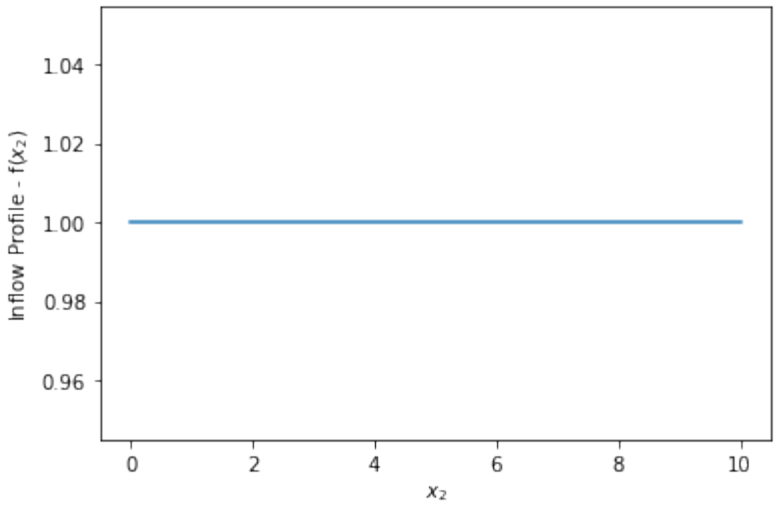
Inflow profile: Since the boundary conditions for velocity at the top and at the bottom are free-slip in this case (`ncly1=nclyn=1`), the inflow profile for streamwise velocity is just 1 everywhere:

[10]:
```
fun = xr.ones_like(ds.y)

# This attribute will be shown in the figure
```

(continues on next page)

(continued from previous page)

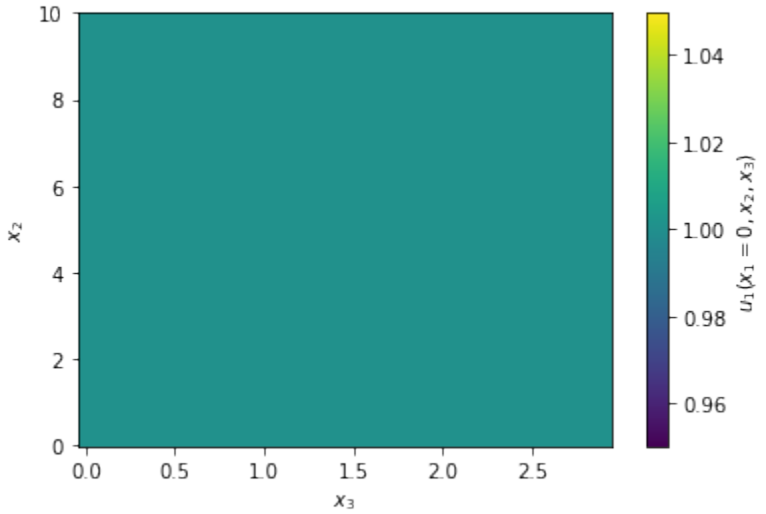
```
fun.attrs["long_name"] = r"Inflow Profile - f($x_2$)"

fun.plot();
```



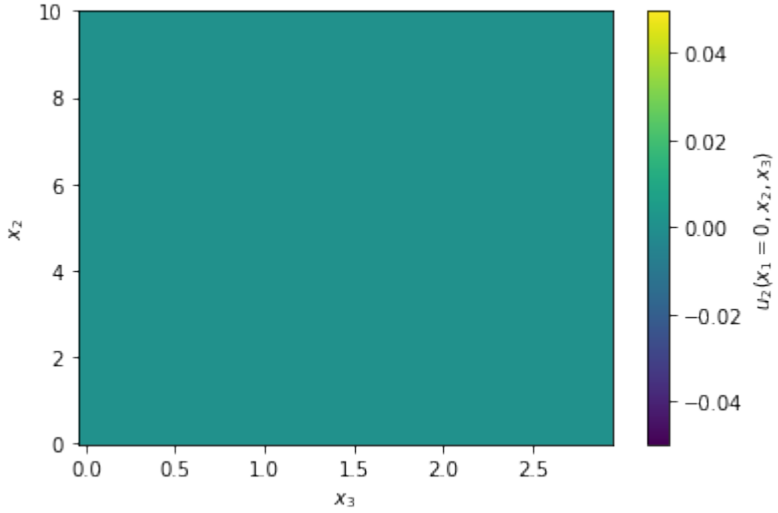
Now, we reset the inflow planes `ds[key] *= 0.0`, just to guarantee consistency in case of multiple executions of this cell. Notice that `ds[key] = 0.0` may overwrite all the metadata contained in the array, so it should be avoided. Then, we add the inflow profile to the streamwise componente and plot them for reference:

[11]:
```
for key in "bxx1 bxy1 bxz1".split():
    #
    print(ds[key].attrs["name"])
    #
    ds[key] *= 0.0
    #
    if key == "bxx1":
        ds[key] += fun
    #
    ds[key].plot()
    plt.show()

plt.close("all")
```

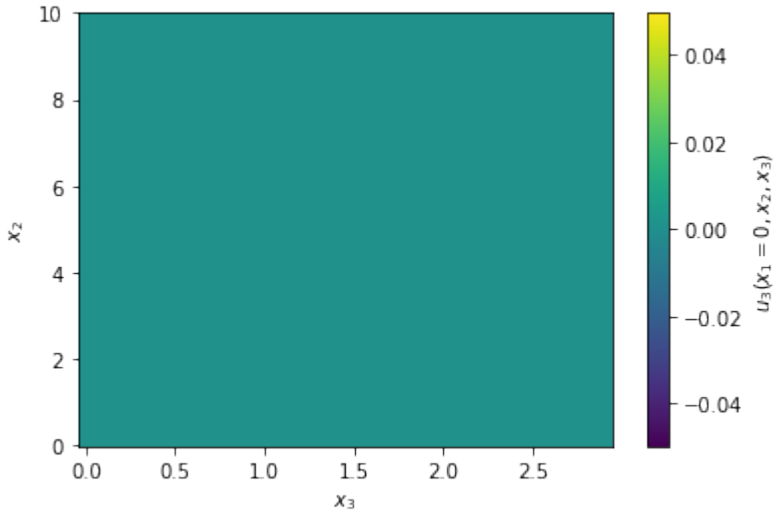
```
Inflow Plane for Streamwise Velocity
```



Inflow Plane for Vertical Velocity



Inflow Plane for Spanwise Velocity

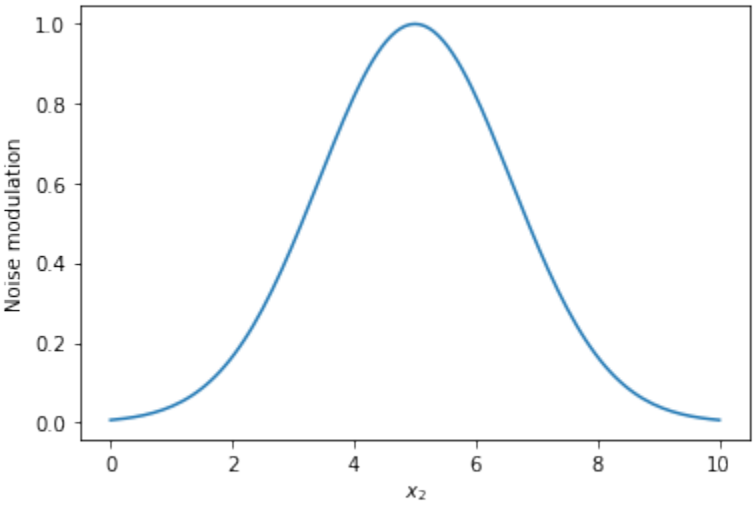


A random noise will be applied at the inflow boundary, we can create a modulation function `mod` to control were it will be applied. In this case, we will concentrate the noise near the center region and make it zero were $y = 0$ and $y = L_y$. The domain is periodic in z `nclz1=nclzn=0`, so there is no need to make `mod` functions of z. The functions looks like:

$$\text{mod} = \exp\left(-0.2(y - 0.5L_y)^2\right).$$

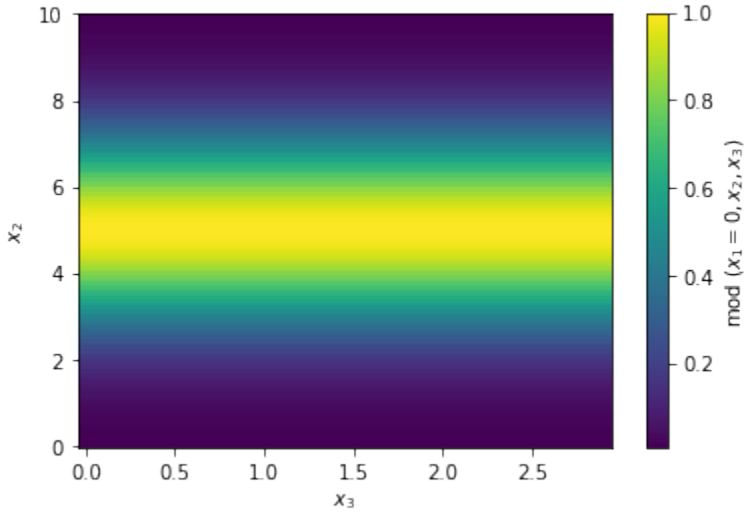
See the code:

[12]:
```
# Random noise with fixed seed,
# important for reproducibility, development and debugging
if prm.iin == 2:
    np.random.seed(seed=67)

mod = np.exp(-0.2 * (ds.y - ds.y[-1] * 0.5) ** 2.0)

# This attribute will be shown in the figure
mod.attrs["long_name"] = "Noise modulation"

mod.plot();
```



Again, we reset the array `ds['noise_mod_x1'] *= 0.0`, just to guarantee consistency in case of multiple executions of this cell. Notice that `ds['noise_mod_x1'] *= 0.0` may overwrite all the metadata contained in the array, so it should be avoided. Then, we add the modulation profile to the proper array and plot it for reference:

[13]:
```
ds["noise_mod_x1"] *= 0.0
ds["noise_mod_x1"] += mod

ds.noise_mod_x1.plot();
```



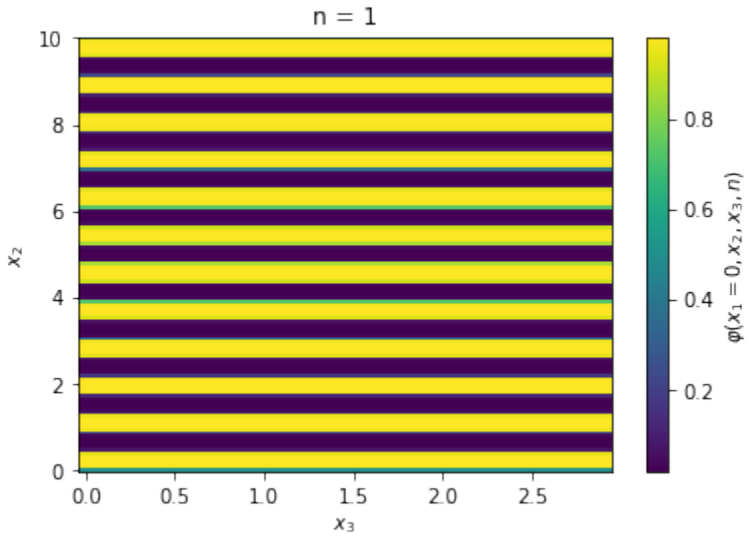
Notice one of the many advantages of using xarray, `mod`, with shape (`ny`), was automatically broadcasted for every point in `z` into `ds.noise_mod_x1`, with shape (`ny`, `nz`).

Inflow BC for the passive scalar: For this case, the choise was a "smooth" square wave, because it is differentiable.

Notice that Xcompact3d supports multiple scalar fields (controlled by `numscalar`, this example just includes one), so different visualization patterns can be set for each one of them.

[14]:
```
# Concentration

print(ds["bxphi1"].attrs["name"])

ds["bxphi1"] *= 0.0

for n in range(prm.numscalar):

    ds["bxphi1"][dict(n=n)] += (
        0.5
        + np.arctan(
            np.sin(2.0 * np.pi * ds.y / prm.yly * 11.5) * (prm.sc[n] * prm.re) ** 0.5
        )
        / np.pi
    )

    ds.bxphi1.isel(n=n).plot()
    plt.show()

plt.close("all")
```

```
Inflow Plane for Scalar field(s)
```



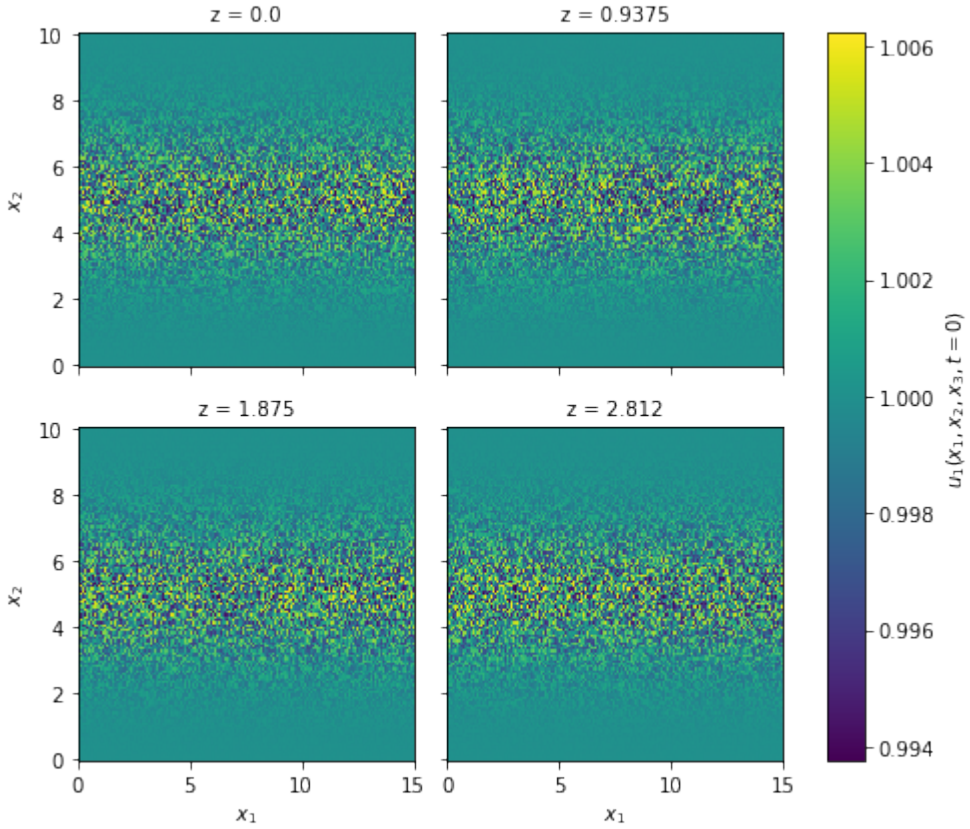
Initial Condition

Now we reset velocity fields `ds[key] *= 0.0`, just to guarantee consistency in the case of multiple executions of this cell.

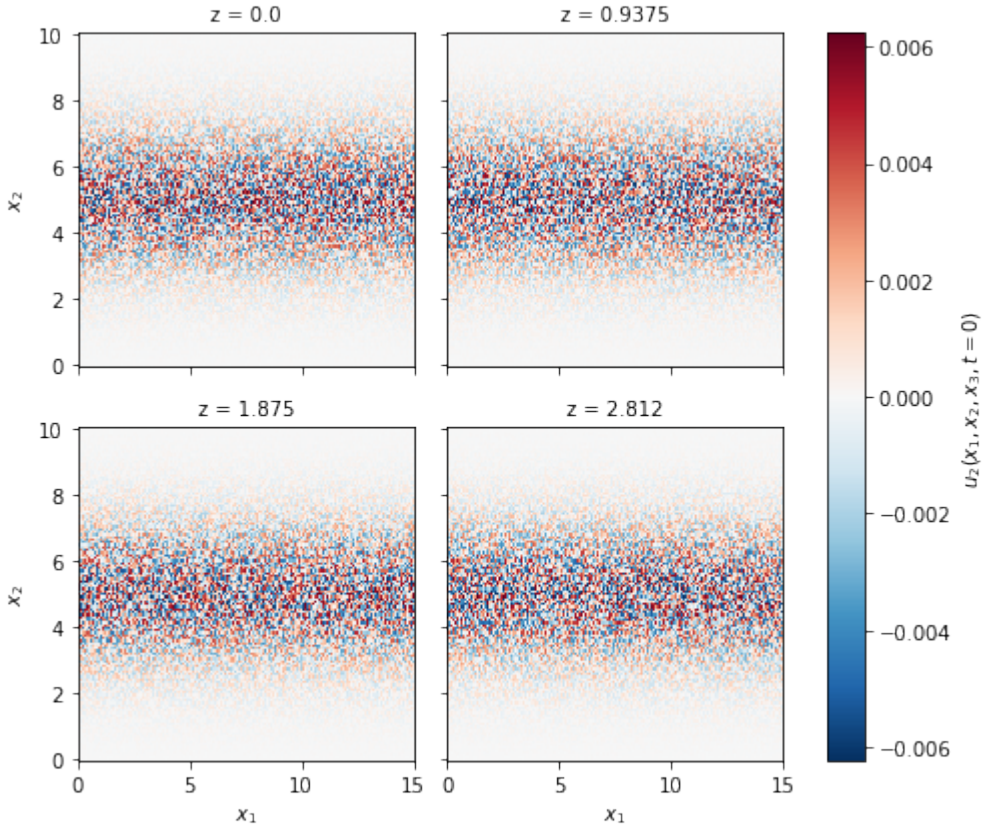
We then add a random number array with the right shape, multiply by the noise amplitude at the initial condition `init_noise` and multiply again by our modulation function `mod`, defined previously. Finally, we add the streamwise profile `fun` to `ux` and make the plots for reference, I'm adding extra options just to exemplify how easily we can slice the spanwise coordinate and produce multiple plots:

[15]:
```
for key in "ux uy uz".split():
    #
    print(ds[key].attrs["name"])
    #
    ds[key] *= 0.0
    ds[key] += prm.init_noise * ((np.random.random(ds[key].shape) - 0.5))
    ds[key] *= mod
    #
    if key == "ux":
        ds[key] += fun
    #
    ds[key].sel(z=slice(None, None, ds.z.size // 3)).plot(
        x="x", y="y", col="z", col_wrap=2
    )
    plt.show()
    #

plt.close("all")
```

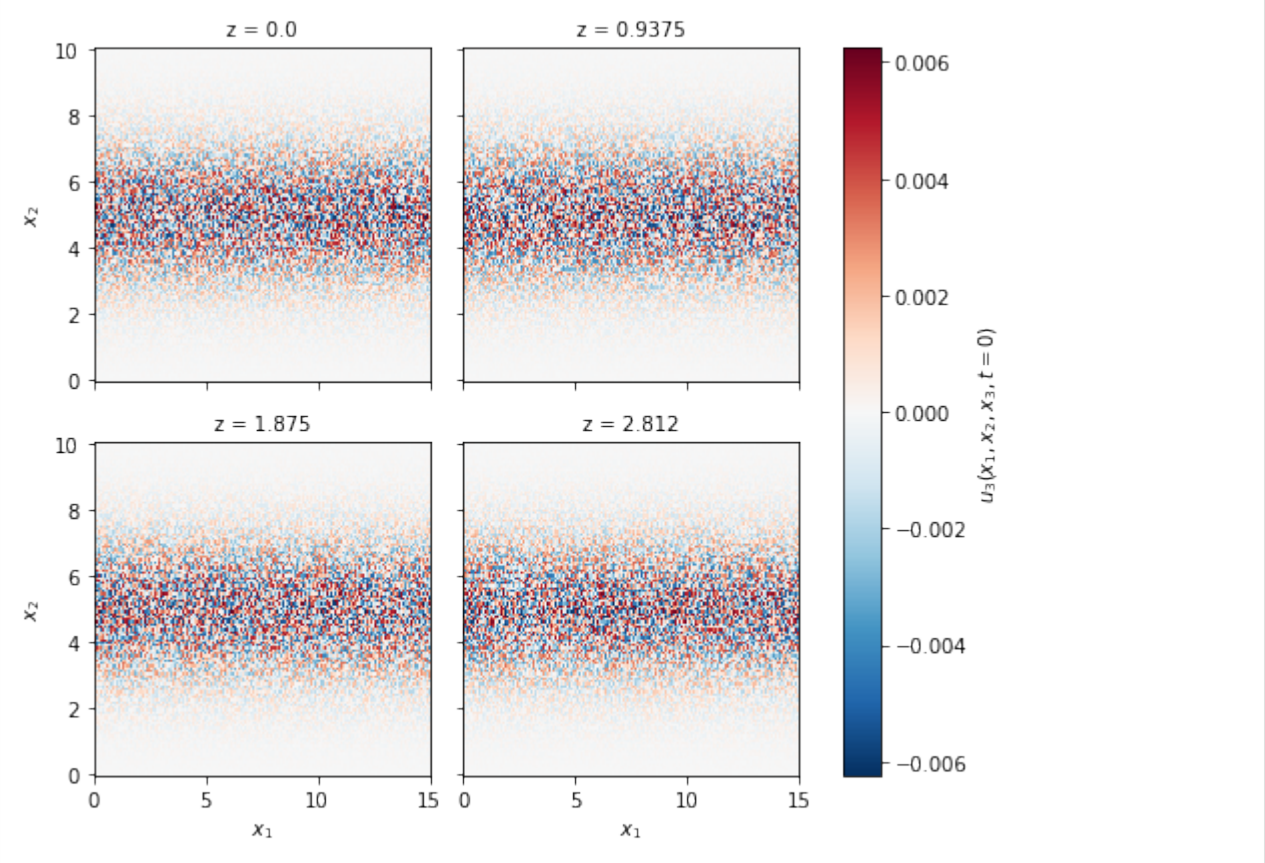
```
Initial Condition for Streamwise Velocity
```



Initial Condition for Vertical Velocity



Initial Condition for Spanwise Velocity



For concentration, let's start with a clean domain:

[16]:
```
ds["phi"] *= 0
```

4.3.3.3 Writing to the disc

is as simple as:

[17]:
```
prm.dataset.write(ds)
```

[18]:
```
prm.write()
```

4.3.3.4 Running the Simulation

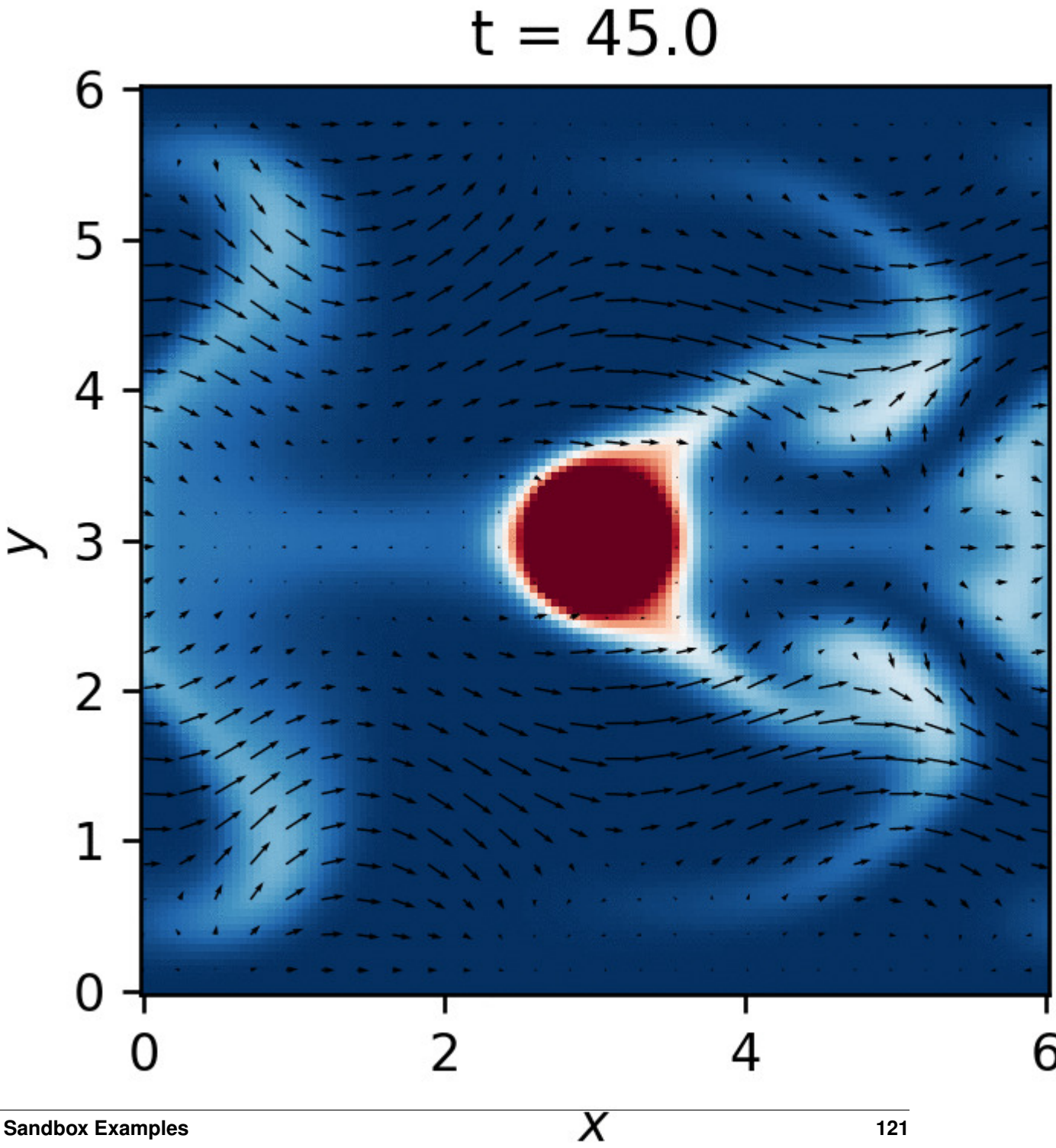
It was just to show the capabilities of `xcompact3d_toolbox.sandbox`, keep in mind the aspects of numerical stability of our Navier-Stokes solver. **It is up to the user to find the right set of numerical and physical parameters**.

Make sure that the compiling flags and options at `Makefile` are what you expect. Then, compile the main code at the root folder with `make`.

And finally, we are good to go:

```
mpirun -n [number of cores] ./xcompact3d |tee log.out
```

4.3.4 Periodic Heat Exchanger



[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import xarray as xr

import xcompact3d_toolbox as x3d
```

4.3.4.1 Parameters

- Numerical precision

Use `np.float64` if Xcompact3d was compiled with the flag `-DDOUBLE_PREC`, use `np.float32` otherwise.

[2]:
```
x3d.param["mytype"] = np.float64
```

- Xcompact3d's parameters

For more information about them, checkout the API reference.

[3]:
```
prm = x3d.Parameters(
    filename="input.i3d",
    # BasicParam
    itype=12,
    p_row=0,
    p_col=0,
    nx=128,
    ny=129,
    nz=8,
    xlx=6.0,
    yly=6.0,
    zlz=0.375,
    nclx1=0,
    nclxn=0,
    ncly1=2,
    nclyn=2,
    nclz1=0,
    nclzn=0,
    iin=1,
    re=300.0,
    init_noise=0.0125,
    inflow_noise=0.0,
    dt=0.0025,
    ifirst=1,
    ilast=90000,
    ilesmod=1,
    iibm=2,
    gravx=0.0,
    gravy=-1.0,
    gravz=0.0,
    # NumOptions
    nu0nu=4.0,
    cnu=0.44,
    # InOutParam
    irestart=0,
    icheckpoint=45000,
    ioutput=500,
    iprocessing=100,
    # LESModel
```

(continues on next page)

(continued from previous page)

```
    jles=4,
    # ScalarParam
    numscalar=1,
    nclxS1=0,
    nclxSn=0,
    nclyS1=2,
    nclySn=2,
    nclzS1=0,
    nclzSn=0,
    sc=[1.0],
    ri=[-0.25],
    uset=[0.0],
    cp=[1.0],
)
```

4.3.4.2 Setup

Geometry

Everything needed is in one dictionary of Arrays (see API reference):

[4]:
```
epsi = x3d.init_epsi(prm)
```

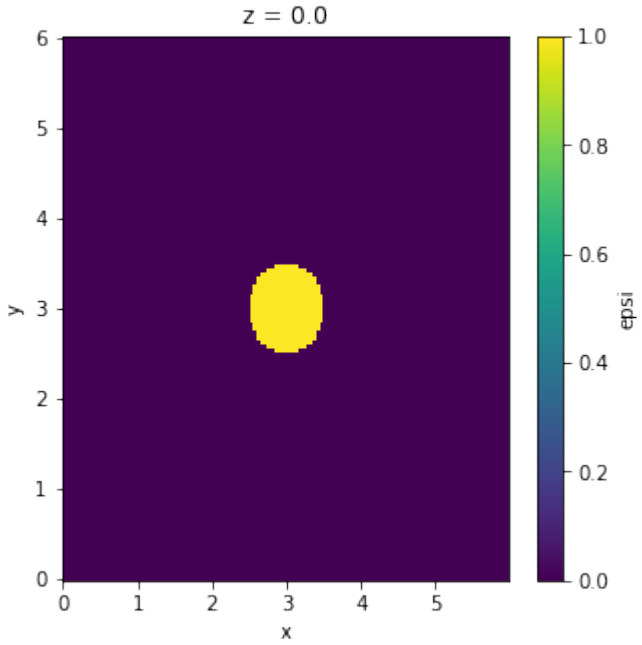
The four ϵ matrices are stored in a dictionary:

[5]:
```
epsi.keys()
```

[5]:
```
dict_keys(['epsi', 'xepsi', 'yepsi', 'zepsi'])
```

Now we draw a cylinder:

[6]:
```
# And apply geo.cylinder over the four arrays
for key in epsi.keys():
    epsi[key] = epsi[key].geo.cylinder(x=prm.xlx / 2.0, y=prm.yly / 2.0,)

# A quickie plot for reference
epsi["epsi"].sel(z=0, method="nearest").plot(x="x", aspect=1, size=5);
```



The next step is to produce all the auxiliary files describing the geometry, so then Xcompact3d can read them:

[7]:
```
%%time
dataset = x3d.gene_epsi_3D(epsi, prm)

prm.nobjmax = dataset.obj.size

dataset
```

```
x
       nobjraf : 1
    nobjmaxraf : 1
           bug : 0

y
       nobjraf : 1
    nobjmaxraf : 1
           bug : 0

z
       nobjraf : 1
    nobjmaxraf : 1
           bug : 0

number of points with potential problem in x : 0
number of points with potential problem in y : 0
number of points with potential problem in z : 0

Writing...
Wall time: 4.52 s
```

[7]:
```
<xarray.Dataset>
Dimensions:       (obj_aux: 2, obj: 1, x: 128, y: 129, z: 8)
Coordinates:
```

(continues on next page)

(continued from previous page)

```
  * obj_aux       (obj_aux) int32 -1 0
  * obj           (obj) int32 0
  * x             (x) float64 0.0 0.04688 0.09375 0.1406 ... 5.859 5.906 5.953
  * y             (y) float64 0.0 0.04688 0.09375 0.1406 ... 5.906 5.953 6.0
  * z             (z) float64 0.0 0.04688 0.09375 ... 0.2344 0.2812 0.3281
Data variables: (12/28)
    epsi          (x, y, z) bool False False False False ... False False False
    nobj_x        (y, z) int64 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0
    nobjmax_x     int64 1
    nobjraf_x     (y, z) int64 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0 0 0
    nobjmaxraf_x  int64 1
    ibug_x        int64 0
    ...            ...
    nxipif_y      (x, z, obj_aux) int64 2 2 2 2 2 2 2 2 2 ... 2 2 2 2 2 2 2 2 2
    nxfpif_y      (x, z, obj_aux) int64 128 2 128 2 128 2 ... 128 2 128 2 128 2
    xi_z          (x, y, obj) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    xf_z          (x, y, obj) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0
    nxipif_z      (x, y, obj_aux) int64 2 2 2 2 2 2 2 2 2 ... 2 2 2 2 2 2 2 2 2
    nxfpif_z      (x, y, obj_aux) int64 7 2 7 2 7 2 7 2 7 ... 2 7 2 7 2 7 2 7 2
```

Boundary Condition

Everything needed is in one Dataset (see API reference):

[8]:
```
ds = x3d.init_dataset(prm)
```

Let's see it, data and attributes are attached, try to interact with the icons:

[9]:
```
ds
```

[9]:
```
<xarray.Dataset>
Dimensions:  (x: 128, y: 129, z: 8, n: 1)
Coordinates:
  * x        (x) float64 0.0 0.04688 0.09375 0.1406 ... 5.812 5.859 5.906 5.953
  * y        (y) float64 0.0 0.04688 0.09375 0.1406 ... 5.859 5.906 5.953 6.0
  * z        (z) float64 0.0 0.04688 0.09375 0.1406 0.1875 0.2344 0.2812 0.3281
  * n        (n) int32 1
Data variables:
    byphi1   (n, x, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    byphin   (n, x, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    ux       (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    uy       (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    uz       (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    phi      (n, x, y, z) float64 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
    vol_frc  (x, y, z) float64 0.0 0.0 0.0 0.0 0.0 0.0 ... 0.0 0.0 0.0 0.0 0.0
```

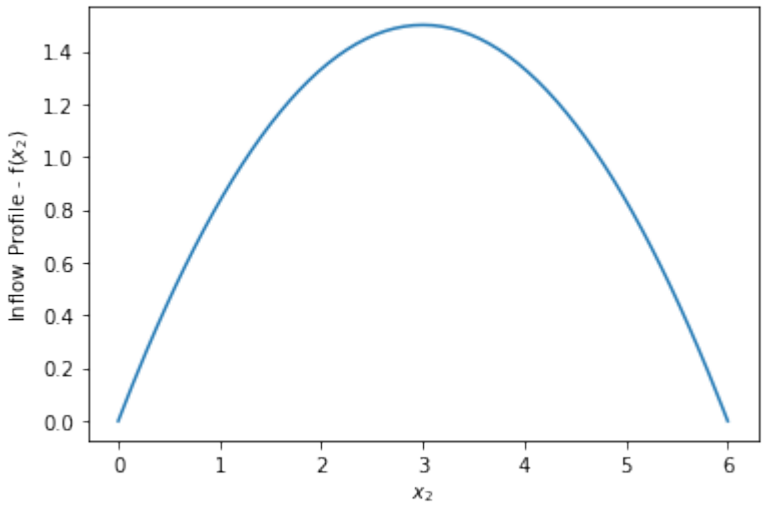
The dimensionless temperature at the objects's surface will always be set to zero by the Immersed Boundary Method, so in opposition to that, let's set the dimensionless temperature at the top and bottom boundaries to one:

[10]:
```
for var in "byphi1 byphin".split():
    ds[var] *= 0.0
    ds[var] += 1.0
```

Initial Condition

Velocity profile: Since the boundary conditions for velocity at the top and at the bottom are no-slip in this case (`ncly2=nclyn=2`), the inflow profile for streamwise velocity must be zero near walls:

[11]:
```
# This function gives the shape
fun = -((ds.y - prm.yly / 2.0) ** 2.0)

# This attribute will be shown in the figure
fun.attrs["long_name"] = r"Inflow Profile - f($x_2$)"

# Now, let's adjust its magnitude
fun -= fun.isel(y=0)
fun /= fun.x3d.simps("y") / prm.yly

fun.plot();
```



Now, let's make sure that the dimensionless averaged velocity is unitary:

[12]:
```
fun.x3d.simps("y") / prm.yly
```

[12]:
```
<xarray.DataArray 'y' ()>
array(1.)
```

A random noise will be applied at the three velocity components, we can create a modulation function `mod` to control were it will be applied. In this case, we will concentrate the noise near the center region and make it zero were $y = 0$ and $y = L_y$. The domain is periodic in z `nclz1=nclzn=0`, so there is no need to make `mod` functions of z. The functions looks like:

$$\text{mod} = \exp\left(-0.5(y - 0.5L_y)^2\right).$$

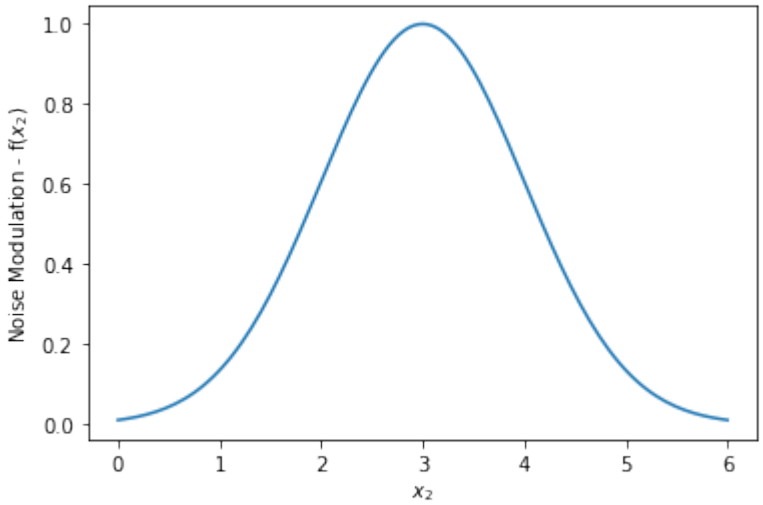
See the code:

[13]:
```
# Random noise with fixed seed,
# important for reproducibility, development and debugging
if prm.iin == 2:
    np.random.seed(seed=67)

mod = np.exp(-0.5 * (ds.y - prm.yly * 0.5) ** 2.0)
```

(continues on next page)

(continued from previous page)

```
# This attribute will be shown in the figure
mod.attrs["long_name"] = r"Noise Modulation - f($x_2$)"

mod.plot();
```

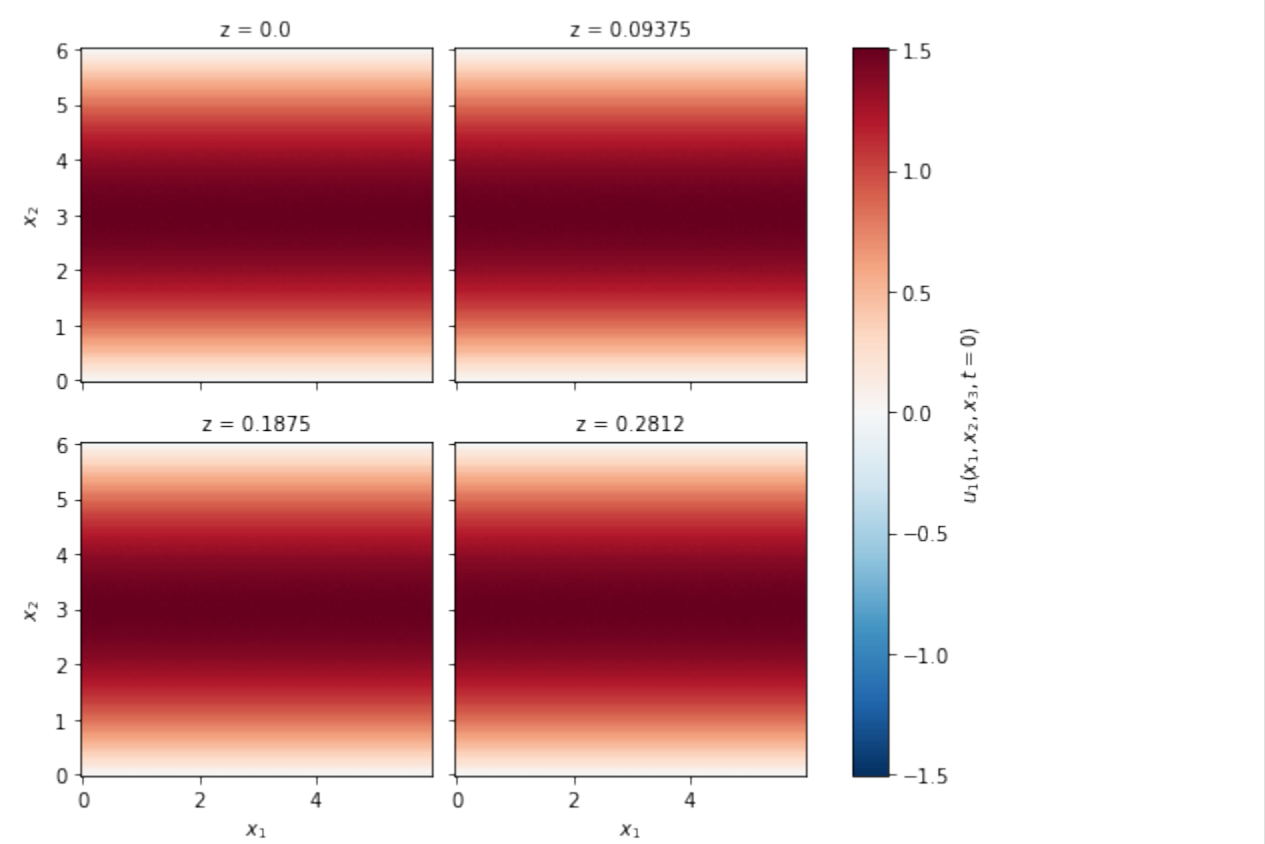


Now we reset velocity fields `ds[key] *= 0.0`, just to guarantee consistency in the case of multiple executions of this cell.

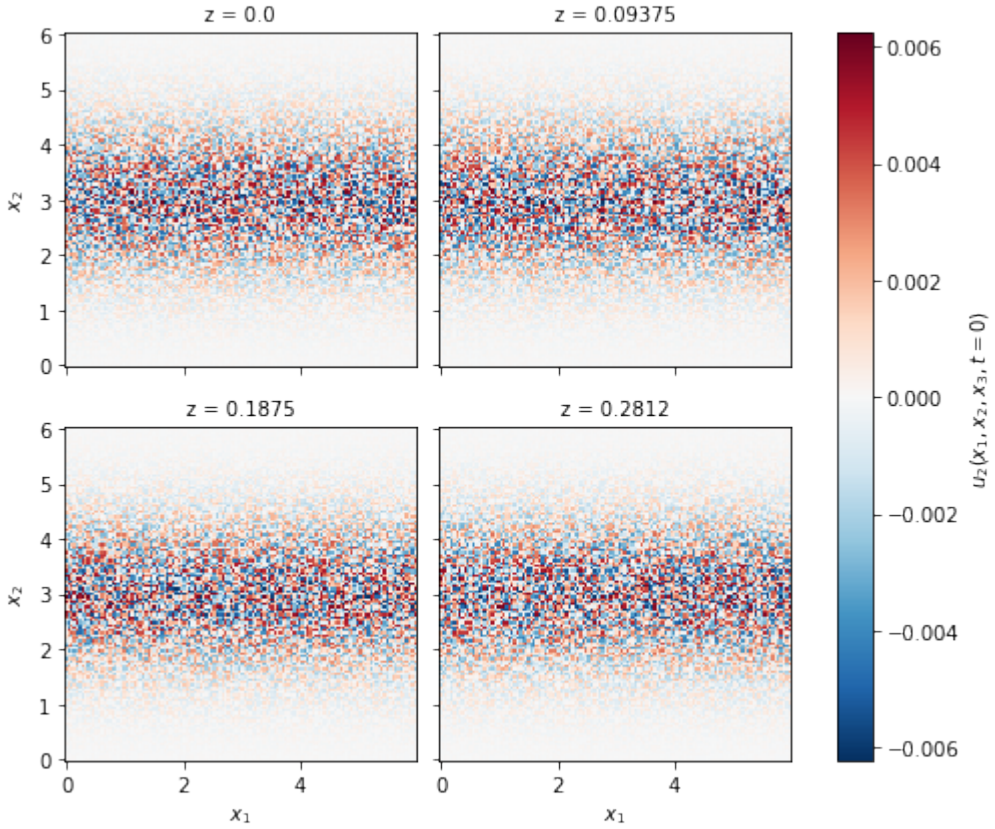
We then add a random number array with the right shape, multiply by the noise amplitude at the initial condition `init_noise` and multiply again by our modulation function `mod`, defined previously. Finally, we add the streamwise profile `fun` to `ux` and make the plots for reference, I'm adding extra options just to exemplify how easily we can slice the spanwise coordinate and produce multiple plots:

[14]:
```
for key in "ux uy uz".split():
    #
    print(ds[key].attrs["name"])
    #
    ds[key] *= 0.0
    ds[key] += prm.init_noise * ((np.random.random(ds[key].shape) - 0.5))
    ds[key] *= mod
    #
    if key == "ux":
        ds[key] += fun
    #
    ds[key].sel(z=slice(None, None, ds.z.size // 3)).plot(
        x="x", y="y", col="z", col_wrap=2, aspect=1, size=3
    )
    plt.show()
    #

plt.close("all")
```

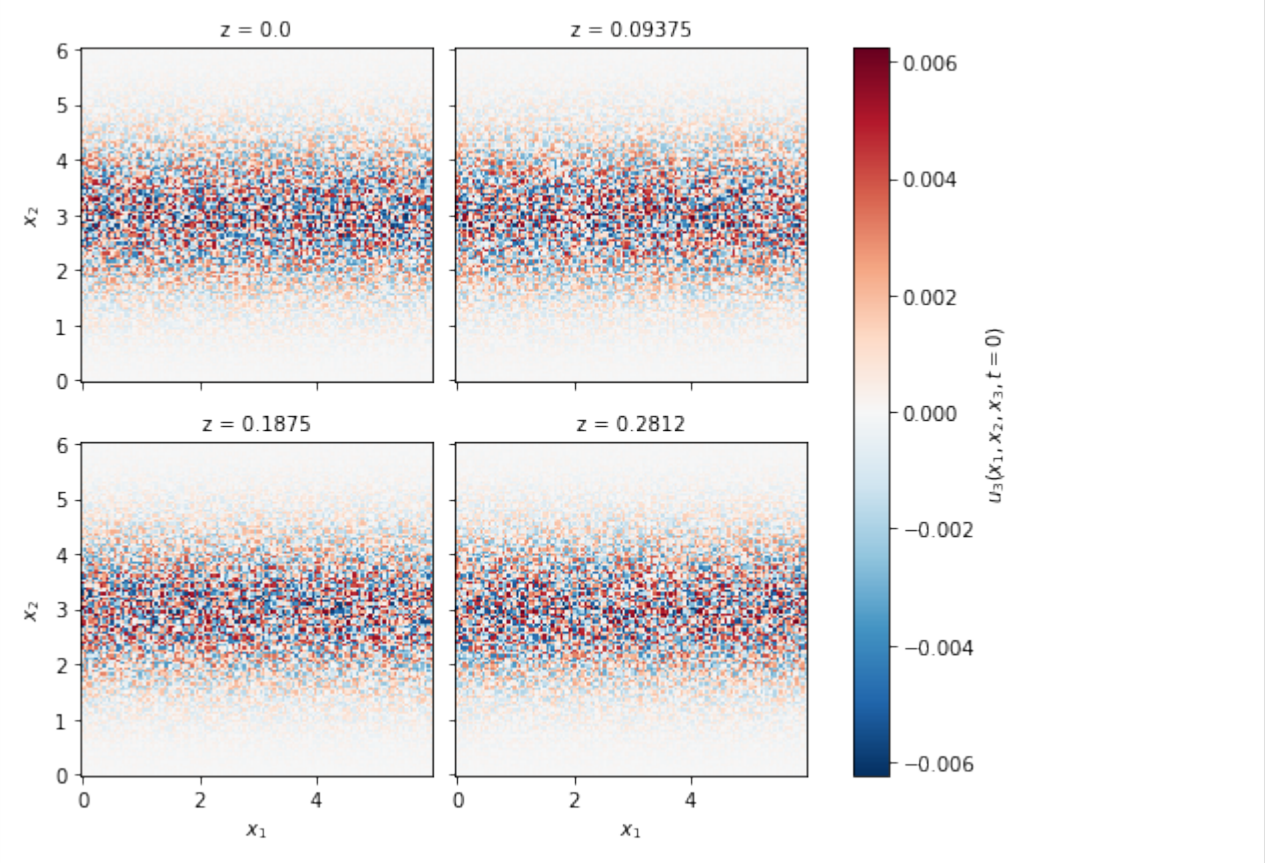
```
Initial Condition for Streamwise Velocity
```



Initial Condition for Vertical Velocity



Initial Condition for Spanwise Velocity



For temperature, let's start with one everywhere:

[15]:
```
ds["phi"] *= 0.0
ds["phi"] += 1.0
```

4.3.4.3 Flow rate control

The *Sandbox flow configuration* is prepared with a forcing term when the flow is periodic in the streamwise direction x_1, in order to compensate viscous dissipation and keep the flow rate constant.

It is done with the help of a personalized volumetric integral operator (`vol_frc`), then, the streamwise velocity will be corrected at each simulated time-step as:

```
I = sum(vol_frc * ux)

ux = ux / I
```

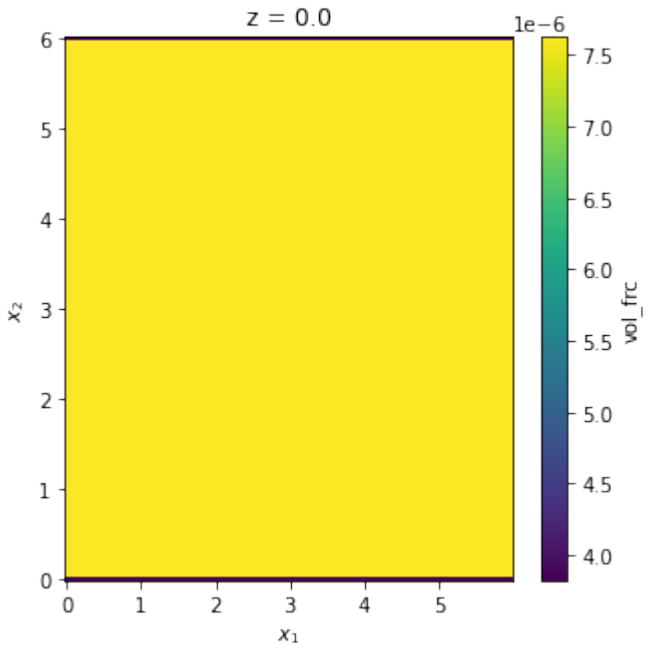
For the composed trapezoidal rule in a uniform grid, the integral operator in the vertical direction can be computed as:

$$vol_{frc} = \Delta y \, [1/2, 1, \ldots, 1, \ldots, 1, 1/2]$$

For a unitary averaged velocity, `vol_frc` must be divided by the domain's heigh. Besides that, for streamwise and spanwise averaged values, `vol_frc` must be divided by `nx` and `nz`.

Finally, `vol_frc` can be coded as:

[16]:
```
ds["vol_frc"] *= 0.0

ds["vol_frc"] += prm.dy / prm.yly / prm.nx / prm.nz

ds["vol_frc"][dict(y=0)] *= 0.5
ds["vol_frc"][dict(y=-1)] *= 0.5

ds.vol_frc.isel(z=0).plot(x="x", y="y", aspect=1, size=5);
```



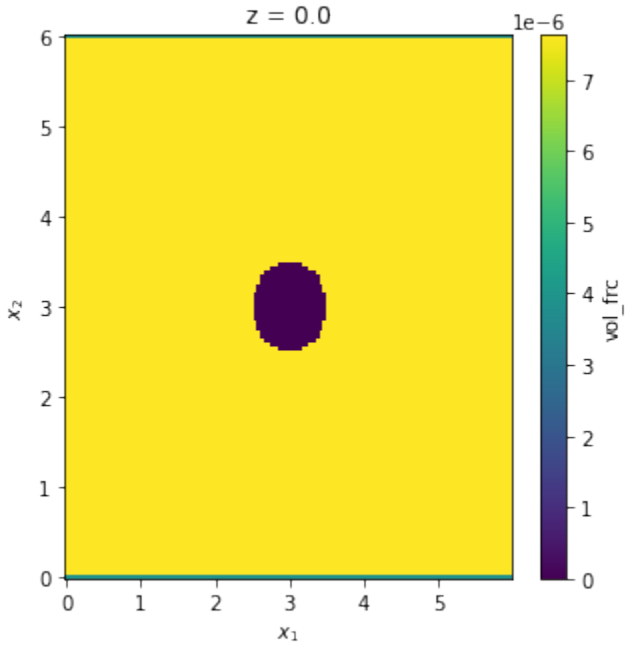
Now, let's make sure that `sum(vol_frc * ux)` is near to one:

[17]:
```
(ds.vol_frc * ds.ux).sum(["x", "y", "z"])
```

[17]:
```
<xarray.DataArray ()>
array(0.99994064)
```

And the last step, we can remove the solid geometry from our integral operator using the code:

[18]:
```
ds["vol_frc"] = ds.vol_frc.where(epsi["epsi"] == False, 0.0)

ds.vol_frc.isel(z=0).plot(x="x", y="y", aspect=1, size=5);
```



4.3.4.4 Writing to the disc

is as simple as:

[19]:
```
prm.dataset.write(ds)
```

[20]:
```
prm.write()
```

4.3.4.5 Running the Simulation

It was just to show the capabilities of `xcompact3d_toolbox.sandbox`, keep in mind the aspects of numerical stability of our Navier-Stokes solver. **It is up to the user to find the right set of numerical and physical parameters**.

The Sandbox Flow Configuration is still in prerelease, it can be found at fschuch/Xcompact3d.

Make sure that the compiling flags and options at `Makefile` are what you expect. Then, compile the main code at the root folder with `make`.

And finally, we are good to go:

```
mpirun -n [number of cores] ./xcompact3d |tee log.out
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

x

Index

Symbols

A

B

C

D

F

G

I

R

S

U

V

W

X

Y

Z