LADiM Documentation

Release 1.1.0

Bjørn Ådlandsvik <bjorn@imr.no>
Institute of Marine Research

2022-06-21

Contents:

CHAPTER 1

Introduction

LADiM is the acronym of the Lagrangian Advection and DIffusion Model.

It is an offline ocean particle tracking model. Presently it takes input from the ROMS ocean model.

This is the third incarnation of the LADiM model at the Institute of Marine Research. The first worked with a version of the Feltfile format used with the Princeton Ocean Model. The second version targeted NetCDF output from the ROMS model. Both these versions were written in Fortran. The new code is written in python (python 3 only). Parts of the code may in future be rewritten in other languages for increased performance.

The old fortran codes had drifted apart without any kind of version control and minimal documentation. Maintenance became increasingly more difficult. The rationale for a new incarnation is to have a well-documented model system under version control. The design should be modular so it can easily be extended to other ocean models and applications.

The model code is available under the MIT license, and is hosted on github, https://github.com/bjornaa/ladim. This documentation is hosted on Read the Docs. A pdf version is also available.

CHAPTER 2

User Manual

2.1 Installation

If you are lucky LADiM is already installed at your system. This can be tested by writing **ladim -h** on the command line. If it is installed, you will get some help text. If it is not installed, or the system version is old, you can make a private install under your user.

TODO: Implement version information as `ladim --version`

2.1.1 Private LADiM installation

If you do not have system-wide permissions you can install LADiM under your own user.

First make sure that you are using python 3.6 or more, by typing **python**:

```
Python 3.6.5 |Anaconda custom (64-bit)| (default, Mar 29 2018, 18:21:58)
[GCC 7.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The python version is the first number. If it is 2.7.x or less you are running legacy python. Try the command **python3**, or on an anaconda system you can change to version 3.6 by `source activate py36` or similar.

If you do not have python 3.6 on your machine, the anaconda distribution is recommended. |

LADiM is hosted on github, download by the command:

```
git clone https://github.com/bjornaa/ladim1.git
```

if you don't have **git** installed, download and unzip the zip-file from the LADiM site above.

This makes a `ladim` directory under the present directory.

Now install LADiM locally, user = your login name:

```
cd ladim
python setup.py install --prefix=/home/user
```

Make sure `/home/user/bin` is in your `PATH`. If you want to override a system LADiM it has to go before python's own bin directory (check: *which python[3]*). Also add `/home/user/lib/python3.x/site-packages` to the environment variable `PYTHONPATH`, where *x* is the minor python version.

And you are ready to try out LADiM.

2.2 Running LADiM

2.2.1 Trying LADiM

With LADiM installed, it is time to test it out.

Go to the examples directory, first `examples/data` and download an example data file `ocean_avg_0014.nc` by the command:

```
python download.py
```

Change to one of the other example directories, for instance `examples/line` and run:

```
python make_release.py
ladim
```

Look at the results by:

```
python animate.py
```

Note: Depending on your python installation you may have to substitute **`python3`** for **`python`** in the examples above.

After browsing through the configuration and particle release chapters below, you can copy the example directory anywhere and modify with other release scenarios or perhaps a ROMS' history or average file of your own.

2.2.2 The main ladim script

Installing LADiM puts the main **`ladim`** script on the PATH. It provides the command:

```
ladim [-h] [-d] [-s] [config_file]
```

The options are:

`-h, --help`
Show a help message and exit

`-d, --debug`
Show more logging information

`-s, --silent`
Show less logging information

`config_file`
Name of optional configuration file, default = `ladim1.yaml`

2.2.3 Running LADiM from python

LADiM can be run inside a python program. This is done by:

```
import ladim
#  ... more lines ...
with open('ladim1.yaml') as fid:
    ladim1.main(config_stream=fid)
```

The main LADiM script, `scripts/ladim`, uses this approach. The `jupyter` example shows how to use this with a triple quoted text string for the configuration, using the standard module `StringIO`.

2.3 LADiM configuration

The LADiM model system is highly configurable using a separate configuration file. The goal is that a user should not have to touch the code, every necessary aspect should be customizable by the configuration file.

The name of the configuration file is given as a command line argument to the main ladim script. If the name is missing, a default name of `ladim1.yaml` is supposed.

It is a goal to provide sane defaults in the configuration file, so that enhancements of the configuration setup do not break existing configuration files.

The configuration file format is a subset of yaml (YAML Ain't Markup Language). Knowledge of `yaml` is not necessary. The example configuration files are self-describing and can easily be modified.

Note: The indentation in a yaml file is mandatory, it is part of the syntax. In particular note that the indentation has to be done by **spaces**, **not tabs**. Reasonable editors will recognize yaml-files by the extension and will automatically produce spaces when you hit the tab key.

See also:

Module *`configuration`* Documentation of the *`configuration`* module

2.3.1 An example configuration file

Below is an example configuration file, `models/salmon_lice/ladim1.yaml`.

2.4 Particle release

Particle release is controlled by a release file. This is a text file with one row per release location per time. The name of the release file is defined in the configuration file.

The format of the line is specified in the configuration by the list `release_format`. If the type of the field is not the default `float`, it should be specified under the `particle_release` heading in the configuration file. Time is indicated by the type code `time`.

There are seven reserved column names.

mult Number of particles. Optional, default = 1.

release_time Time of particle release with format `yyyy-mm-ddThh:mm:ss` or `"yyyy-mm-dd hh:mm:ss"`. Mandatory.

X Grid X-coordinate of the release position

Y Grid Y-coordinate of the release position

lon Longitude of release position

lat Latitute of release position

Z Release depth in meters (positive downwards). Mandatory

Note: It is essential that the release time is interpreted as a *single string*. This can be done by a "T" between date and clock parts or by enclosing it with double ticks. Time components can be dropped from the end with obvious defaults.

Note: Either (X, Y) or (lon, lat) must be given, If both are present, the grid position (X, Y) is used.

Note: Using (lon, lat) requires a `ll2xy` method in the *gridforce.Grid*.

Additional variables should be specified in the configuration file. If their type is not `float`, the type (`int`, `bool`, `time`) should be specified. [How about strings?]

Example:

Using the configuration:

```
particle_release:
    # release_type, discrete or continuous
    release_type: continuous
    release_frequency: [1, h]   # Hourly release
    variables: [mult, release_time, X, Y, Z, farmid, super]
    particle_variables: [release_time, farmid]
    # Converters (if not float)
    mult: int
    release_time: time   # np.datetime64[s]
    farmid: int
```

A typical line in the release file may look like this:

```
5 2015-07-06T12 379.12 539.23 5.0 10041 1243.2
```

This means that 5 particles are released at time `2015-07-06 12:00:00`, at a location with `farmid`-label 10041, with grid coordinates (379.12, 539.23) at 5 meters depth. Each particle is a superindividual with weight 1243.2 so the particle release corresponds to a total of 6216 individuals. The release is repeated every hour until a later particle release anywhere (or the end of the simulation).

Warning: Strange things may happen if particle release is not aligned with the model time stepping. The user is presently responsible for synchronizing model and release times.

Note: Entries with release_time before the start or after the stop time of LADiM are ignored. In particular, constant continuous release will not work if the release_time is before the model's start time. (still true?)

Note: Entries in a release row after the configured fields are silently ignored. (at least should be, check).

Note: Particles released outside the model grid are ignored. A warning is logged.

Note: Particles released on land are retained in the simulation, but do not move. In particular check that particles released by longitude and latitude near the coast is in a sea cell. TODO: Provide a warning for particles initially on land.

See also:

Module `release` Documentation of the `release` module

2.5 Output

The particle distributions are written to NetCDF files. The file name, output frequency, included variables and their attributes are governed by the configuration file.

2.5.1 Output format

This is a not backwards compatible modification of the old LADiM format. The fundamental data structure is the same, so scripts should be easy to modify. The change is done to follow the CF-standard as far as possible, and to increase flexibility.

For particle tracking, the CF-standard defines a format for "Indexed ragged array representation of trajectories". This is not suitable for our purpose since we are more interested in the geographical distribution of particles at a given time than the individual trajectories. Chris Baker at NOAA has an interesting discussion on this topic and a suggestion at github. The new LADiM format is closely related to this suggestion.

The format uses an indexed ragged array representation, suitable for both version 3 and 4 of the NetCDF standard. The dimensions are `time`, `particle` and `particle_instance`. The particle dimension indexes the individual particles, while the particle_instance indexes the instances i.e. a particle at a given time. The number of times is defined by the length of the simulation and the frequency of output, both are determined by the configuration. The number of particles is given by the sum of the multiplicities in the particle release file, also known in advance. The number of instances is more uncertain as particles may be removed from the computation by leaving the area, stranding, or becoming inactive for biological reasons. The particle_instance dimension is therefore the single unlimited dimension allowed in the NetCDF 3 format.

The indirect indexing is given by the variable `particle_count(time)`. The particle distribution at timestep `n` (counting from initial time n=0) can be retrieved by the following python code:

```python
nc = Dataset(particle_file)
particle_count = nc.variables['particle_count'][:n+1]
start = np.sum(particle_count[:n])
count = particle_count[n]
X = nc.variables['X'][start:start+count]
```

Note that some languages uses 1-based indexing (MATLAB, fortran by default). In MATLAB the code is:

```
% Should be checked by someone who knows matlab
particle_count = ncread(particle_file, 'particle_count', 1, n+1)
start = 1 + sum(particle_count(1:n))
count = particle_count(n+1)
X = ncread(particle_file, 'X', start, count)
```

2.5.2 Particle identifier

The particle identifier, `pid` should always be present in the output file. It is a particle number, starting with 0 and increasing as the particles are released. The `pid` follows the particle and is not reused if the particle becomes inactive. In particular, `max(pid) + 1` is the total number of particles involved so far in the simulation and may be larger than the number of active particles at any given time. It also has the property that if a particle is released before another particle, it has lower `pid`.

The particle identifiers at a given time frame has the following properties,

- `pid[start:start+count]` is a sorted integer array.
- `pid[start+p] >= p` with equality if and only if all earlier particles are alive at the time frame.

These properties can be used to extract the individual trajectories, if needed.

2.5.3 Example CDL

NetCDF has a text representation, Common Data Language, abbreviated as CDL. Here is an example CDL, produced by **`ncdump -h`**:

```
netcdf out {
dimensions:
      particle = 72000 ;
      particle_instance = UNLIMITED ; // (540000 currently)
      time = 13 ;

variables:
      double time(time) ;
            time:long_name = "time" ;
            time:standard_name = "time" ;
            time:units = "seconds since 2015-04-01T00:00:00.000000" ;
      long particle_count(time) ;
            particle_count:long_name = "number of particles in a given timestep" ;
            particle_count:ragged_row_count = "particle count at nth timestep" ;
      double release_time(particle) ;
            release_time:units = "seconds since 2015-04-01T00:00:00" ;
            release_time:long_name = "particle release time" ;
      long farmid(particle) ;
            farmid:long_name = "fish farm location number" ;
      long pid(particle_instance) ;
            pid:long_name = "particle identifier" ;
      float X(particle_instance) ;
            X:long_name = "particle X-coordinate" ;
      float Y(particle_instance) ;
            Y:long_name = "particle Y-coordinate" ;
      float Z(particle_instance) ;
            Z:standard_name = "depth_below_surface" ;
            Z:positive = "down" ;
```

(continues on next page)

(continued from previous page)

```
            Z:units = "m" ;
            Z:long_name = "particle depth" ;
      float super(particle_instance) ;
            super:long_name = "number of individuals in instance" ;
      float age(particle_instance) ;
            age:standard_name = "integral_of_sea_water_temperature_wrt_time" ;
            age:units = "Celcius days" ;
            age:long_name = "particle age in degree-days" ;

// global attributes:
      :Conventions = "CF-1.5" ;
      :institution = "Institute of Marine Research" ;
      :source = "Lagrangian Advection and Diffusion Model, python version" ;
      :history = "Created by pyladim" ;
      :date = "2017-02-15" ;
}
```

2.5.4 Split output

For long simulations, the output file may become large and difficult to handle. Version 1.1 adds the possibility of split output. This is activated by adding the keyword `numrec` to the output section in the configuration file. This will split the output file after every numrec record. If the `output_file` has the value `out.nc`, the actual files are named `out_0000,nc`, `out_0001.nc`, `...` .

2.5.5 Restart

Version 1.1 also adds the of warm start, most typically a restart. In this mode LADiM starts with an existing particle distribution, the last time instance in an output NetCDF file.

This mode is triggered by the specification of a `warm_start_file` in the configuration. As the initial distribution already exist on file, it is by default not rewritten.

Restart and split output works nicely together. Suppose `numrec` is present and equal in both config files. If the warm start file is nam ed `out_0030.nc` and is complete, the new files will start at `out_0031.nc` and continue as in the original simulation. With no diffusion and unchanged settings, the new files should be identical to the original.

See also:

Module *`output`* Documentation of the *`output`* module.

2.6 Workflow

This chapter describes various possible workflows with LADiM, from simple usage to developing. Some terminology may be useful. An *installed* LADiM, has been installed by `python setup.py install` in the LADiM root directory, either *system-wide* by sudo or personal by using the `--prefix`-option, see *Installation*. This is typical a working *stable* version in a usable state.

2.6.1 Unmodified use

The simplest workflow is to simply use the installed LADiM. Use a work directory, copy and modify a `ladim1.yaml` file to point to your input files and adjust the simulation time and other settings. Run the simulation by the command ladim1.

The configuration file can be renamed and saved to make the model simulation reproducible. To run LADiM with a named configuration file, give the file name as a command line argument to the ladim command.

Possible complication: if you are in a python2 environment ,,,

2.6.2 Modify the IBM

The IBM modules can be very different, with very configuration requirements. The configuration system is therefore not able to configure all details of the IBMs. Consequently, IBMs are the most likely modules where the code has to be modified.

To modify an IBM, copy the file to your working directory. Modify the `ibm -> ibm-module` setting in the configuration file to point to the local module. Thereafter, the installed ladim command should work properly.

A totally new IBM, for another species perhaps, can be developed locally in the work directory and activated in the same way. The *IBM* module description documents what an *IBM* class should make available. An example of this is the minimal IBM in the `examples/killer` directory. As the new IBM matures, it should be included in the `ladim/ibms` directory.

2.6.3 Modify other modules

The code of other modules may be modified in a slightly different way. Copy the module to the working directory and modify it as above. To activate it, copy the main ladim script `script/ladim` to the working directory and modify the import statement to point to the local module. To make sure that the correct script is executed, run the simulation with the command `./ladim`.

A common use case may be to develop new *gridforce* modules to work with output from different ocean models or idealized settings. The *gridforce* description specifies the necessary public attributes and methods for the `Grid` and `Forcing` classes. The logo example, `examples/logo` has a minimal idealized *gridforce*-module. In future, the LADiM distribution may contain a collection of different gridforce modules and the choice be available by the configure mechanism in the same manner as for the *IBM*.

The trick of using a local `ladim`-script may be use for other modifications of as well.

2.6.4 Maintenance and further development

For develop work on LADiM it is important to not mess up the stable installed version. This can be done by virtualization. Use a separate `git` branch for the development and a separate `conda` or `virtualenv` environment. The conda solution is described below.

The `git` part is set up by:

```
git clone https://github.com/bjornaa/ladim1.git
cd ladim
git branch mydevelop
git checkout mydevelop
```

Use `conda env list` to find the environments, find one with python 3, say `py36` and clone it locally, for instance in `$HOME/conda/envs`:

```
conda create  -p $HOME/myenv --clone py36
source activate $HOME/myenv
python setup.py install
```

This installs the developing version in the `myenv`-environment. Do a `python setup install` after every change before running LADiM. To get back to the stable environment do:

```
git add ...
git commit ...
git checkout master
source deactivate
```

Instead of `source deactivate` it is enough to simply kill the working window. The next time, it is easier. It is enough to write:

```
git checkout mydevelop
source activate $HOME/myenv
```

If the development is a general improvement or important addition to the standard LADiM a pull request should be sent to github so that it can be included in the stable version.

2.7 Post processing

LADiM comes with a simple python package `postladim` that can be used for visualisation and analysis of LADiM output. It is based on the `xarray` package http://xarray.pydata.org/en/stable/.

2.7.1 Usage

The basic class is `ParticleFile` giving access to the content of a LADiM output file.

```
from postladim import ParticleFile
pf = ParticleFile("output.nc")
```

The `ds` method shows the underlying xarray Dataset, which is useful for a quick overview of the content and for more advenced data processing.

The time at a given time step n is given by:

```
pf.time[n]
```

The output may be a bit verbose, `pf.time(n)` is syntactic sugar for the more concise‘‘pf.time[n].values‘‘.

The number of particles at time step n is given by:

```
pf.count[n]
```

For use in indexing `pf.count` is an integer array instead of an xarray DataArray. The DataArray is avaiable as `pf.ds.particle_count`.

Following pandas and xarray, an instance variable, like X, is given both by attribute `pf.X` and item `pf['X']` notation. The NetCDF inspired notation `pf.variables['X']` is obsolete.

The most basic operation for an instance variable is to get the values at time step n as a xarray DataArray:

```
pf.X[n]
```

An alternative notation is `pf.X.isel(time=n)`. The time stamp can be used instead of the time index:

```
pf.X.sel(time='2020-02-05 12')
```

The format is optimized for particle distributions at a given time. Trajectories and other time series for a given particle may take longer time to extract. For the particle with identifier *pid=p*, the X-coordinate of the trajectory is given by:

```
pf.X.sel(pid=p)
```

If many trajectories are needed, it may be useful to turn the dataset into a full (i.e. non-sparse) 2D DataArray, indexed by time and particle identifier.

```
pf.X.full()
```

Note that for long simulations with particles of limited life span, this array may become much larger than the Particle-File.

2.7.2 Reference

ParticleFile

class ParticleFile(*particle_file*)

Class for LADiM result files.

ds
The underlying xarray Dataset.

num_times
Number of time frames in the file.

num_particles

The number of distinct particles

count
Integer numpy ndarray of particle counts at different time steps

start
ndarray of start indices at different time step

end
ndarray of end indices, short hand for pf.start + pf.count

instance_variables
List of particle instance variables

particle_variables
List of particle variables.

time
xarray DataArray of time stamps

position(*n*)
Tuple with position (X, Y) of particle-distribution at n-th time time, `pf.position(n) = (pf.X[n], pf.Y[n])`

trajectory(*pid*)
Returns a tuple of X and Y coordinates of the particle with identifier pid, `trajectory(pid) = (pf.X.sel(pid=pid), pf.Y.sel(pid=pid))`.

variables
Deprecated, dictionary of variables, `pf.variables['X'] = pf['X'] = pf.X`.

time(*n*)
Syntactic sugar, `pf.time(n) = pf.time[n].values`

particle_count(*n*)
Deprecated, `pf.particle_count(n) = pf.count[n]`

InstanceVariable

class InstanceVariable

da
The underlying xarray DataArray

count
Number of particles at given time

time
The time steps

num_particles
The number of distinct particles

isel(*time=n*)
Select by time index, `V.isel(time=n) = V[n]`.

sel(*time=t*, *pid=p*)
Select by time value and/or pid value

`V.sel(time='2019-09-19 12')` selects particles at that time value `V.sel(pid=23)` selects the time history of particle with that pid value `V.sel(time='2019-09-19 12', pid=23)` selects the unique particle instance

full()
Return the full (non-sparse) 2D DataArray. May become vary large.

The InstanceVariable class support item notation:

```
V[n] = V.isel(time=n)
V[n, p] = V.isel(time=n).sel(pid=p)
```

Note that `V[n, p]` may be different from `V[n][p]` if particles have been removed.

The *length* of an instance variable is the number of time steps, `len(V) == len(V.time)`.

ParticleVariable

class ParticleVariable

This class represents a variable that depends on the particle, but is independent of time.

da

Underlying xarray DataArray

Supports item notation:

```
``V[p]`` is value of particle with ``pid = p``.
```

2.8 Examples

LADiM comes with a set of examples that try to illustrate different features. They are available in the directory `ladim/examples`. Most examples use an example NetCDF file `ocean_avg_0014.nc` from ROMS to be put in `ladim/examples/data`. This directory contains a script `download.py` that downloads the file from an ftp server at IMR.

Most of the examples are executed by the command sequence:

```
python make_release.py
ladim
python animate.py
```

Alphabetic list of examples:

`jupyter` The jupyter example shows how to wrap a LADiM session in a jupyter notebook, and how to animate the model results. It is a remake of the `line` example.

`killer` This is a modified version of the `streak` example. It demonstrates how to include a simple Individual Based Model (IBM). Particles are released continuously from a point and are “killed” at the age of two days.

`latlon` The latlon example shows how to release particles at positions given in geographical coordinates and how to get longitude and latitude into the output file. It also shows how to visualize the results on a map using the mapping extensions `basemap` and `cartopy` to `matplotlib`.

`line` This is the basic example. Particles are released instantaneously from a line across the North Sea from Scotland to Norway. The particle distribution is followed for a period twelve days.

`logo` This example generates the LADiM logo. The text LADiM is written with particles in the North Sea and deformed by the current for six days.

`nested` This examples shows how use two nested model grids and let particles move between them. This is done by a separate `gridforce` module that calls the gridforce for each of the two grids.

`obstacle` This example shows how to include your own gridforce module, and how to run an analytically defined simulation. Here, flow around a semi-circular peninsula (classical inviscid flow around a cylinder).

`restart` This example shows how to split the LADiM results into files of limited number of time frames, and how to restart a LADiM run from an output file.

`station` This example demonstrates transport from a single location at different depths.

`streak` This example demonstrates a streak line, continuous release of particles from a fixed location.

CHAPTER 3

LADiM Modules

This chapter describes briefly the present implementation of LADiM. The code is modular and flexible with classes that can be modified or replaced for different purposes.

The descriptions tries the requirements of the classes from the rest of the system, hopefully being helpful for developing alternative classes for use systems.

The code may be developed towards base classes, where alternatives can inherit from a common base class. Presently, alternatives must be written separately.

3.1 `main` — Main LADiM Module

This module contains one function, *`main()`* governing LADiM simulations. Its signature is:

`main.`**`main`**(*config_stream*, *loglevel=logging.INFO*)

Parameters

- **`config_stream`** (*`stream`*) – Configuration stream
- **`loglevel`** (*`int`*) – Logging level, default = logging.INFO

The configuration stream is normally an open yaml-file, but can be a text string by `StringIO` as in the jupyter example.

A simplified, but working, version of the **`ladim`** script:

```
import logging
from ladim import main

loglevel = logging.INFO
config_file = 'ladim1.yaml'

logging.basicConfig(
    level=loglevel, format='%(levelname)s:%(module)s - %(message)s')
```

(continues on next page)

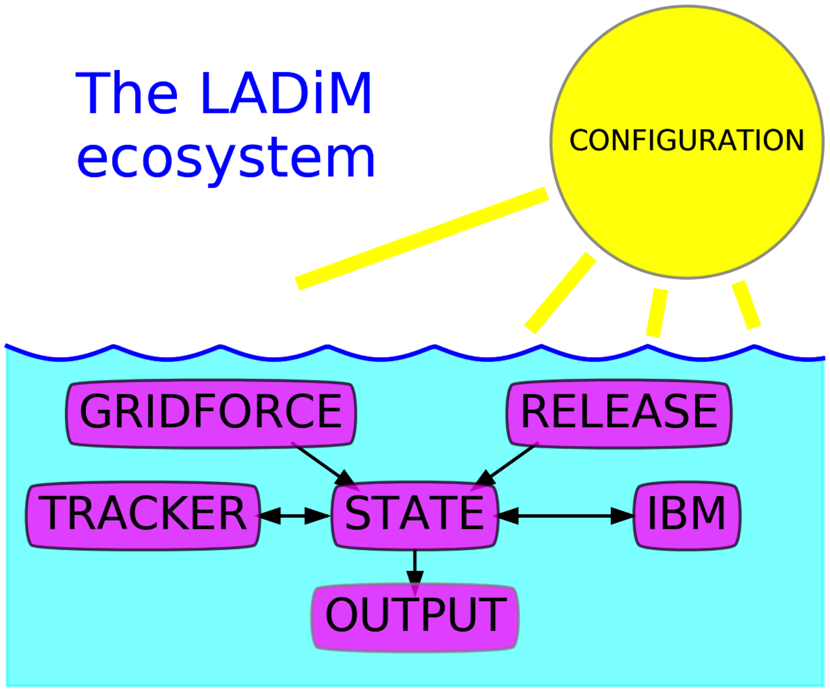

Fig. 1: The LADiM “eco”-system; the connections between the modules.

(continued from previous page)

```
with open(config_file, encoding='utf8') as fp:
    main(config_stream=fp, loglevel=loglevel)
```

3.2 `configuration` — LADiM Configuration

LADiM's configuration system uses the `pyyaml` package to read the configuration file. This allows comments, nested keywords, and flexibility in the sequence of keywords, missing or extra keyword,

The configuration procedure makes a dictionary. It does not quite match the structure of the yaml configuration file as some of the values are derived or default. Presently the dictionary is somewhat inconsistent to provide backwards compatibility. Future versions will continue to separate the configuration info into separate directories for the gridforce, ibm and output modules.

start_time Start time for simulation, [numpy.datetime64]

stop_time Stop time for simulation, [numpy.datetime64]

reference_time Reference time for simulation, [numpy.datetime64] Used in the units attribute in the netCDF output file

particle_release_file Name of particle release file

output_file Name of output file or template for sequence of output files

start Simulation start "cold" or "warm"

warm_start_file Name of warm start file (if needed)

dt Model time step in seconds [int]

simulation_time Simulation tile in seconds [int]

numsteps Number of time steps [int]

gridforce Gridforce module with configuration Dictionary of information to the gridforce module

input_file Name of input file or template for sequence of input files

ibm_forcing List of extra forcing variables beside velocity

ibm IBM module with configuration

ibm_variables: List of variables needed by the IBM module

ibm_module Path to the IBM module

release_type Type of particle release, "discrete" or "continuous".

release_format List of variables provided during particle release

release_dtype Dictionary with name -> type for the release variables

particle_variables Names of particle variables among the release variables

output_format NetCDF format for the output file

skip_initial Logical switch for skipping output of intitial field

output_numrec Number of time records per output file, zero means no output splitting

output_period Hours between output [int]

num_output Number of output time records

output_particle Particle variables included in output

output_instance Instance variables included in output

nc_attributes mapping: variable -> dictionary of netcdf attributes

advection Advection scheme, "EF" = Euler Forward, "RK2" = Runge-Kutta order 2, "RK4" = Runge-Kutta order 4

diffusion Logical switch for horizontal random walk diffusion

diffusion_coefficient Diffusion coefficient, constant [m/s**2]

3.3 `gridforce` — Ladim Module for Grid and Forcing

This module describes the grid used for particle tracking and interfaces the forcing from the ocean model. This is the only module in LADiM that depends on the model. Interfacing other models is done by adding an alternative gridforce module. The module `gridforce_ROMS` implements a ROMS interface. The obstacle example shows how to make a simple gridforce module in idealized cases.

3.3.1 Coordinate system in LADiM

The coordinate system should be orthogonal, and may be curvilinear. Grids as used by ROMS for instance are suitable.

The grid spacing is given by horizontal arrays of metric coefficients Δx and Δy.

The vertical coordinate system is simply the depth in meters, with positive values downwards.

3.3.2 `Grid`

class `gridforce.`**`Grid`**(*config*)

The argument config is a `Configuration` object

The grid is defined by the Grid class having the following public attributes:

`imax`

The number of grid cells in X-direction.

`jmax`

The number of grid cells in Y direction.

And a set of methods, taking the particle positions as input:

`metric`(*X*, *Y*)

Returns Δx and Δy at the positions.

`sample_depth`(*X*, *Y*)

Returns the bottom depth at the positions.

`lonlat`(*X*, *Y*)

Returns longitude and latitude at the positions [degrees].

`ingrid`(*X*, *Y*)

Check if the particles are inside the grid.

`atsea`(*X*, *Y*)

Check if the particles are at sea (not on land).

3.3.3 `Forcing`

The *`Forcing`* class provides the current from the ocean model simulation that drives the particle tracking. It also includes extra fields necessary for the (biological) behaviour (the IBMs).

class `gridforce.`**`Forcing`**`(`*config*, *grid*`)`

The arguments config and grid are `Configuration` and *`Grid`* objects.

A forcing object should have the following public attributes and methods:

`step`

List of timesteps where forcing is available

`velocity``(`*X*, *Y*, *Z*, *tstep=0*`)`

Returns velocity (U, V) in the particle positions. tstep is a fraction of the time step ahead, 0 for present, 1 for next timestep, 0.5 for half way between.

`field``(`*X*, *Y*, *Z*, *name*`)`

Sample a field, name is the name of the field in the state and ibm modules Examples: 'temp', 'salt', . . .

`W``(`*X*, *Y*, *Z*`)`

Vertical velocity at particle positions [Not implemented yet]

3.4 `release` — Particle release

During initialization, the *`ParticleRelease`* class has the responsibility to warn about any mismatches in timing between the simulation and particle release.

class `release.`**`ParticleRelease`**`(`*config*`)`

Public attributes are: [need only one of these]

`steps`

List of time steps with particle release.

`total_particle_count`

The total number of particles in the simulation

`particle_variables`

Dictionary with particle variables and their values

It is implemented as an iterator. The `__next__()` method, returns a dictionary of release information to be used by the State class' append method.

3.5 `state` — The model state

This module keeps track of the model state, that is the values of the state variables at a given time.

class `state.`**`State`**`(`*config*`)`

Mandatory state variables:

`pid`

Particle identifier

`X, Y`

Horizontal position [grid coordinates]

`Z`

Vertical position [meters]

IBM state variables may vary and are defined in the configuration. For example, for salmon lice these are `age` - the "age" in degree-days, and `super` - the number of individuals represented by a super-individual. These extra attributes can be accessed through their name as `state['age']` for instance.

In addition the state keeps record of time:

`timestep`
The time step counter

`timestring`
The present time in ISO 8601 format: "yyyy-mm-dd hh:mm:ss".

The State class has methods:

`update`(*grid*, *forcing*)
Advance the model to the next step.

`append`(*new*)
Append new particles to the state, new is a dictionary of new particles and their state.

3.6 `tracker` — Particle tracking

This is the physical particle tracking module for horizontal movement. Vertical movement (except staying between bottom and surface) is the responsibility of the IBM class.

`class` `tracker.`**`Tracker`**(*config*)
It is controlled by configuration parameters: `config.dt`, `config.advection`, `config.diffusion`, `config.diffusion_coefficient`

There are no public attributes

One public method:

`move_particles`(*grid*, *forcing*, *state*)

Advect and diffuse the particles horizontally, check for landing or particles out-of-area.

3.7 `IBM` — Individual Based Module

An IBM (Individual Based Model) for biological behaviour or similar can be added to LADiM. As they may be very different, it is difficult to be general enough.

The IBM variables are named by the configuration. Initial values may be provided by the particle release file. Alternatively values may given or derived from the IBM forcing fields. If none of the above, the values are initially set to zero. [Check if correct, may change to error if uninitialized]

The IBM module has the responsibility for updating the required extra forcing (besides velocity) by calling the `forcing.field` method.

A repository of IBM modules are maintained by Pål N. Sævik on github.

Requirements for an IBM module:

`class` `IBM.`**`IBM`**(*config*)

`update_ibm`(*self*, *grid*, *state*, *forcing*)

This method is supposed be called from the `State.update()` method.

3.7.1 IBM support utilities

A `light` module is available as `ladim1.ibms.light`, that can be used to compute the surface light (without clouds). This is useful for modelling diel vertical migration for many species. The module is based on the paper by A. Skartveit & J.A. Olseth (1988). It is given as a function:

`IBM.`**`surface_light`**(*dtime*, *lon*, *lat*)

Parameters

- **`dtime`** (*`numpy.datetime64`*) – Time
- **`lon`** (*`float`*) – Longitude [degrees]
- **`lat`** (*`float`*) – Latitude [degrees]

3.8 `output` — NetCDF output of particle distributions

The *`OutPut`* class is responsible for the NetCDF output from the simulation. Initially it defines the NetCDF output file. It is used sequentially, saving the selected part of the state at regular intervals.

`class` `output.`**`OutPut`**(*config*)

The class has two methods:

`write`(*state*)

Saves the selected state variables.

`write_particle_variables`(*release*)

Writes the selected particle variables.

Here particle variables are things like, `release time`, `release position`, that belongs to a particle and does not change with time.

Note that the `pid` is by definition a particle variable, but it is used as an instance variable to identify the particle of an instance. It works as the index of the particle variables.

CHAPTER 4

LADiM implementation

4.1 Programming language

The LADiM code is written in python, more specific it requires python 3.6 or newer.

4.1.1 Dependencies

In addition to the python standard library, the model is based on the numerical package numpy and the NetCDF library netcdf4-python. The yaml package pyyaml is used for the configuration, while the particle release module depend on the data analysis package pandas. All these packages are available in the anaconda bundle.

The postprocessing library, *postladim*, uses the xarray framework. In addition the examples use the plotting package matplotlib for visualisation.

Testing is done using the pytest-framework.

This documentation uses the Sphinx documentation system to generate html and pdf documentation. To produce the pdf version a decent LaTeX implementation is needed.

4.2 Variables in LADiM

To obtain the necessary flexibility of different IBMs and different gridforce module determined by at run time by the configuration, it is necessary to have a strict terminology for variables.

First some basics, a `particle` is the fundamental entity of particle tracking. It has positions and possibly other attributes and keeps its identity troughout the simulation. A `particle instance` or simply an `instance` is a particle at a fixed time. The `pid`, short for particle identifier, maps a particle instance to its particle.

Particle variable A variable associated to a particle, that does not depend on time.

Particle instance variable A variable associated to a particle instance. Typically time-dependent.

Position variables Instance variables, X, Y, and Z, that gives the position of the instance in grid coordinates.

Particle identifier The projection, `pid` from particle instance to particle. It is considered an instance variable.

Mandatory variables The position variables and the particle identifier

State variables The total collection of variables that determine the model state.

IBM-variable A non-mandatory state variable.

Derived variables Variables such as longitude and latitude, derived from the state variables.

Forcing variable A variable from the forcing file that is used to modify the model state. Main examples are the mandatory horizontal velocity components U and V. Non-mandatory forcing variables, for instance temperature, are called IBM forcing variables.

Release variable A variable read from the particle release file, such as initial position.

Output variable A variable that is written to the output file

Some particle variables, for instance release site code, has no influence on the dynamics and are simply written to output for extra information. Other particle variables may store initial information, like release time or release position, and has no further influence. Other particle variables may influence the dynamics throughout the simulation. Examples are diameter or sinking velocity of non-organic sediment particles. The last category are counted as IBM-variables.

Some variables have predefined names. For the state these are the position variables, `X`, `Y`, `Z`, the particle identifier `pid`, and the logical flags `alive` and `active`. Similarly, the derived longitude and latitude have names `lon` and `lat` respectively. For the forcing variables it is the horizontal velocity components `U` and `V` in the grid directions and the vertical velocity `W` (positive upwards). IBM variables are referenced indirectly like state.temp or state['temp'] where state is a State object.

The particle identifier, `pid` is per definition a particle variable. However, it "lives" on particle instances, identifying the particle the instance belongs to. It is therefore considered an instance variable.

Python Module Index

Index