Overview of the pyladim program

Pyladim is (presently) a pure python particle tracking code for offline use based on output from the ROMS ocean model.

The horizontal coordinates used are ROMS grid coordinates. (continuous X, Y s.t. X = i and Y = j in the center of grid cell (i,j)). The landmask is taken directly from ROMS. In the vertical, depth is used as a coordinate. A full version handle input/output in longitude/latitude.

The data structure is simple. The model state is described by a class *State*, with float32 numpy arrays *X*, *Y*, *Z* for the particle positions, and integer arrays *pid* and *start* for the particle identifier (particle counter) and start time. The forcing is in a class *ROMS_input* which holds and updates the 3D velocity arrays (in native staggered grid with s-coordinates).

The main program is called *ladim*. It starts by reading a configuration or setup file. Thereafter it initalizes the *State*, *ROMS_input*, and *OutPut* instances. The main time loop has the following steps:

```
- Update the forcing
- Add new particles, if needed
- Write to file, if needed
- Move the particles
- Add any behaviour (i.e. vertical migration)
```

Mulig svakhet: Skriver ut initial state OK men ikke final state, må ta et ekstra tidsteg for å få dette med.

The movement is done by functions in *trackpart.py*. In the prototype only the simple Euler-Forward advection method is provided. Higher order advection schemes (Runge Kutta 4th order) and random walk diffusion can easily be added. The routine(s) in trackpart calls sample-functions in *sample_roms.py* to linearly interpolate the fields to the particle positions.

ROMS-avhengigheten er i klassen *ROMS_input* og sample-funksjonene i *sample_roms.py* (de siste burde vært metoder i klassen). For å bruke andre modeller må en lage tilsvarende klasse for hver modell.

Design

A cleaner version of the design (not identical to the actual prototype, not complete) can be summed up by the following classes:

```
class State

particle_identifier
X, Y, Z
```

```
start_time
(all 1D numpy arrays over active particles)

update            # This is the real particle tracking
seed_particles
remove_particles
summary           # returns string with summary information

# --------------------

class Forcing   # with subclasses like ROMS_Forcing

U, V, W, ...
(1D numpy arrays sampled at particle positions)

update
(different update methods for different models
 includes reading files and interpolation)

# -------------------

class Setup

timestep
start_time
stop_time
nsteps      # number of time steps
grid_file
input_file
particle_release_file
output_period
output_file
output_variables  # List

readsup   # read configuration file
writesup

# -------------------------

class Grid  #  with subclasses ROMS_Grid and so on

mask        # sea mask

ll2grid     # lon/lat -> grid coordinates
grid2ll     # grid coordinates -> lon/lat

# -------------------------

class ParticleReleaser
```

```
particle_counter
next_release_step

read_particles     # read time and location of release

# -------------------------

class OutPut

nc_type     # nc_type['X'] = 'f4' etc
nc_attr     # nc_attr['X']['long_name'] = 'grid X-coordinate' etc

write       # write the state
close       # close the output file

# --------------------------

class Ladim  # Main program

setup_file        # Configuration file
timestep_counter

inititiate_state
initiate_forcing
initiate_output   # Define the netCDF file

next      # making it a python iterator, return timestep_counter
__iter__
clean_up     # Final clean-up
```

The main program will be a script that could look something like:

```
model = Ladim(setup_file)

model.initiate_state()  # These could be part of the __init__
model.initiate_forcing()
model.initiate_output()

for stepnr in model:
    model_time = stepnr * model.setup.dt
    if model_time % 86400 == 0:
        # Daily summary
        print model.state.summary()

model.clean_up()
```