OpenDA course and exercises

Nils van Velzen, Martin Verlaan, Stef Hummel

July 14, 2017

Installation of OpenDA

Before you can start with the exercises you must first install OpenDA. For the latest instructions, you are referred to `$OPENDA/doc/OpenDA_domunentation.pdf`, section "Installation" or the same document on our website `www.openda.org`.

1 Exercise 1: Getting started

Edward Lorenz (1963) developed a very simplified model of convection called the Lorenz model. The Lorenz model is defined by three differential equations giving the time evolution of the variables x, y, z:

$$\frac{dx}{dt} = \sigma(y - x) \tag{1}$$

$$\frac{dy}{dt} = \rho x - y - xz \tag{2}$$

$$\frac{dz}{dt} = xy - \beta z \tag{3}$$

where σ is the ratio of the kinematic viscosity divided by the thermal diffusivity, ρ the measure of stability, and β a parameter which depends on the wave number. The implementation of the Lorenz model that is available in OpenDA is solved numerically by using a Runge-Kutta method.

This model, although simple, is very nonlinear and has a chaotic nature. Its solution is very sensitive to the parameters and the initial conditions: a small difference in those values can lead to a very different solution.

The purpose of this exercise is to get you started with OpenDA. You will learn to run a model in OpenDA, make modifications to the input files and plot the results.

- The input for this exercise is located in directory `Exercise_1`. For Linux and Mac OS X, go to this directory and start `oda_run.sh`, the main application of OpenDA. For Windows, start the main application with `oda_run_gui.bat` from the `$OPENDA/bin` directory. The main application allows you to view and edit the OpenDA configuration files, run your simulations and visualize the results.

- Try to run a simulation with the Lorenz model. You can use the configuration file `simulation_unperturbed.oda`. The results are written to

`simulation_unperturbed_results.m`, You can make a plot using Octave or Matlab. Load the results using the function `load_results`:

```
[t,xyz,tobs,obs]=load_results('
simulation_unperturbed_results');
plot3(xyz(1,:),xyz(2,:),xyz(3,:));
grid on;
```

Listing 1: Matlab

Or with python. To run the examples you need python 2.7 with the packages numpy and matplotlib installed. Depending on your environment you may need to import these packages.

```
import numpy as np
import matplotlib.pyplot as plt
```

Listing 2: Python initialize

and next make the plot

```
#load data
import simulation_unperturbed_results as sim
# make 3d line plot
from mpl_toolkits.mplot3d import Axes3D
fig1 = plt.figure()
ax = fig1.add_subplot(111, projection='3d')
# note we start counting at 0 now
Axes3D.plot(ax,sim.x[:,0],sim.x[:,1],sim.x[:,2])
```

Listing 3: Python

Now make a plot of only the first variable of the model (`xyz(1,:)`).

```
plot(t,xyz(1,:),'b')
```

Listing 4: Matlab

```
plt.figure()
plt.plot(sim.model_time,sim.x[:,0],'b')
plt.show()
```

Listing 5: Python

- Observations of the first variable are available as well. Make a plot of the observations together with the simulation results.

```
    [t,xyz,tobs,obs]=load_results('
simulation_unperturbed_results');
    plot(t,xyz(1,:),'b')
    hold on
    plot(tobs,obs,'r*');
    hold off
```

Listing 6: Matlab

```
    import simulation_unperturbed_results as sim
    plt.plot(sim.model_time,sim.x[:,0])
    plt.plot(sim.analysis_time,sim.obs,'r*')
```

Listing 7: Python

- Then you can start an alternative simulation with the lorenz model that starts with a slightly different initial condition using the configuration file `simulation_perturbed.oda` that starts with slightly different initial conditions.
- Visualize the unperturbed and perturbed results in a single plot. Make a 3d trajectory plot and a 2d plot in time of first variable. Do you see the solutions diverging like the theory predicts?

```
    [t1,xyz1,tobs1,obs1]=load_results('
simulation_unperturbed_results');
    [t2,xyz2,tobs2,obs2]=load_results('
simulation_perturbed_results');
    figure(1)
    plot3(xyz1(1,:),xyz1(2,:),xyz1(3,:),'b');
    hold on
    plot3(xyz2(1,:),xyz2(2,:),xyz2(3,:),'r');
    hold off
    legend('unperturbed','perturbed')

    figure(2)
    plot(t1,xyz1(1,:),'b')
    hold on
    plot(t2,xyz2(1,:),'r')
    hold off
    legend('unperturbed','perturbed')
```

Listing 8: Matlab

```
    #load unperturbed and perturbed results
    import simulation_unperturbed_results as sim
    import simulation_perturbed_results as simp
```

```
  fig3 = plt.figure()
  ax = fig3.add_subplot(111, projection='3d')
  Axes3D.plot(ax,sim.x[:,0],sim.x[:,1],sim.x[:,2],'b')
  Axes3D.plot(ax,simp.x[:,0],simp.x[:,1],simp.x[:,2],'r'
)

  fig4 = plt.figure()
  plt.plot(sim.model_time,sim.x[:,0],'b')
  plt.plot(simp.model_time,simp.x[:,0],'r')
```

Listing 9: Python

- Create a modified example that uses an ensemble forecast with perturbed initial conditions. You can do this in a number of steps:
  - Create the input file `simulation_Ens.oda` based on `simulation_unperturbed.oda`. Change the algorithm and the configuration of the algorithm.
    hint: the algorithm is called
    org.openda.algorithms.kalmanFilter.SequentialEnsembleSimulation.
  - Write the configuration file of the Ensemble algorithm (e.g. named `algorithm/EnsSimulation.xml`) with the following content:

```
  <?xml version="1.0" encoding="UTF-8"?>
  <sequentialAlgorithm>
     <analysisTimes type="fromObservationTimes" ></
analysisTimes>
     <ensembleSize>5</ensembleSize>
     <ensembleModel stochParameter="false"
                    stochForcing="false"
                    stochInit="true" />
  </sequentialAlgorithm>
```

Listing 10: XML-input for sequentialAlgorithm

Hint: do not forget to reference `algorithm/EnsSimulation.xml` in `simulation_Ens.oda`.

- Run this ensemble simulation and read the results in Octave or Matlab using `load_ensemble.m` and slightly different for python
  - make a plot of the first variable of the five ensemble members in a single plot

```
  [t,ens]=load_ensemble('simulation_ensemble_results
');
  ens1=reshape(ens(1,:,:),size(ens,2),size(ens,3));
  plot(t,ens1)
```

Listing 11: Matlab

```
import ensemble
import simulation_ensemble_results as res
(t,ens)=ensemble.reshape_ensemble(res)
ens1=ens[:,0,:] #note we start counting at 0
fig5 = plt.figure()
plt.plot(t,ens1)
```

Listing 12: Python

– make a plot of the mean of the first variable

```
plot(t,mean(ens1,2))
```

Listing 13: Matlab

```
fig6 = plt.figure()
plt.plot(t,np.mean(ens1,1))
```

Listing 14: Python

– run the same simulation again[1] but now with an ensemble size of 10, 50, 100 and 200 and plot the mean of the first variable. What do you see, and what does this mean?

2 Exercise 2: Some basic properties of the EnKF

In this exercise you will learn how to set up and run the EnKF method in OpenDA.

- Prepare the input files for a run with the EnKF method. Use the input files from exercise 1 as template. Hint: the Ensemble Kalman filter is called org.openda.algorithms.kalmanFilter.EnKF. The algorithm configuration file has the following content

```
<?xml version="1.0" encoding="UTF-8"?>
<EnkfConfig>
   <ensembleSize>10</ensembleSize>
   <ensembleModel stochParameter="false"
                  stochForcing="false"
                  stochInit="true" />
</EnkfConfig>
```

Listing 15: XML-input for EnKF algorithm

[1] For large models or ensemble sizes a huge amount of output is generated. Your run will be much faster when you disable the messages in the gui, by pressing the 'Disable Logging' button. You can also run without the gui, by using the command `oda_run.sh <inputfile>` (Linux/Mac OS X) or `oda_run_batch.bat <inputfile>` (Windows)

- Plot the ensemble mean of the first model variable and the observations. With some luck the solution should track the observations.
  Tip: use the scripts `load_obs.m` and `load_ensemble.m` for reading the data into matlab (cf. Exercise1), or `load_ensemble.py` for python.

- Look at the observation input file of the StochObserver. The StochObserver does not only describe the observations but the accuracy as well. Can you make a new observation input file with similar observed values but with a 10 times larger standard deviation for the observation error. Tip: you can edit the file in OpenOffice or MS Excel or use the find and replace function of an advanced text editor.

- Repeat the run with EnKF but now for the new observations and plot the first variable of the ensemble means and the observations. What do you see and what is the reason for this behavior of the algorithm?

- The number of ensemble members controls the accuracy of the ensemble approximation. What happens if you increase the number to e.g. 100, or decrease it to 5? Use (initially) observations with a standard deviation of 5.0. Experiment as well with various standard deviations of the observations.

3 Exercise 3: Steady-state

In this section you will learn how to create and use a steady-state Kalman filter with OpenDA. The example model we use in this section is a 1-dimensional wave model:

$$\frac{\partial h}{\partial t} + D\frac{\partial v}{\partial x} = 0 \tag{4}$$

$$\frac{\partial v}{\partial t} + g\frac{\partial h}{\partial x} + c_f v = 0 \tag{5}$$

$$\tag{6}$$

With $h(x,t)$ the (water) level above the reference plane, $v(x,t)$ the velocity, $D(x)$ the depth under the reference plane, g the gravitational acceleration c_f the friction coefficient and $x \in [0, L]$ the location. For our model we have selected the boundary values $v(x = L, t) = 0$ and $h(x = 0, t) = \frac{1}{5}sin(2\pi t)$. An AR(1) model is defined on the left water level boundary.

- Look at the implementation of the model in `WaveStochModelInstance.java`, in the directory `simple_wave_model/java/src/org/openda/`. See how the state is defined and how the model is discretized. If you want you can compile the model using `ant build` as we will explain in excercise 6. However to make it easy for you, you will find the compiled version of this model,`simple_wave_model.jar` in the directory `simple_wave_model/bin`.

- The model represents a "user" model that is not part of the OpenDA distribution. Therefore you have to copy the model jar-file to the bin directory of your OpenDA installation. In this way OpenDA can find this model.

- Run the model (`waveSimulation.oda`) and visualize the model results (`plot_movie.m` or `plot_movie.py`). Do not forget to add the jar-file of the model to the `CLASSPATH` variable, or to copy the jar-file into the bin directory of your OpenDA version
- Adjust the input files in order to run the model with stochastic forcings.
- Generate water level observations from this stochastic run. We need observations at (approximately) $x = \frac{1}{4}L$, $x = \frac{1}{2}L$ and $x = \frac{3}{4}L$. You can use the script `generate_obs.m` for this task. We want to have observations at $t = 0.1, 0.2, ..., 10.0$, (initially) select a standard deviation of 0.05.
- Run the Ensemble Kalman filter (`waveEnkf.oda`). This run will generate and write gain matrices at specified times. Find where and how this is specified in the input.
- Plot the columns of the gain matrices. (The script `plot_gains.m` or `plot_gains.py` plots the water level part of the gain matrices). What do these columns mean?
- (Re)generate the gain matrices using different numbers of ensembles. When you compare the gain matrices, what do you notice. Note: The algorithm will generate an enormous amount of output when you run the EnKF with a very large number of ensembles (e.g. 500). You can suppress the output by commenting out (or remove) the `resultWriter`-part of the oda-input file.
- Use the generated steady state gain matrices for a steady state Kalman run (`waveSteadystate.oda`). Compare the performance of:
  - a (non-stochastic) run without filtering,
  - an EnkF run with various numbers of ensembles (do not forget to reinstate the resultWriter if you have switched it off),
  - the various steady state gains.

  you can use the scripts `plot_obs_sim.m`, `plot_obs_ens.m` and `plot_obs_steady.m` and similar routines for python.
- Generate (observations) gain matrices but now for only a single observation. Make sure that the observed values are exactly the same as in the 3 observation observer. Compare the columns of the 3-observation gain matrices to the single observation matrices. What is the main difference and why?

4 Exercise 4: A black box model - Calibration

A simple way to connect a model to OpenDA is by letting OpenDA access the input and output files of the model. OpenDA cannot directly understand the input and output files of an arbitrary model. Some code has to be written such that the black box model implementation of OpenDA can read and write these files. In this exercise you will learn how to connect an existing model to OpenDA assuming that all the input and output files of the model can indeed

be accessed by OpenDA. The exercise focusses on the configuration of the black box wrapper in OpenDA.

In the directory `exercise_4/original_model/` you will find a model written in python
`reactive_pollution_model.py` and a compiled version of this code for windows. There is also an input file (`reactive_pollution_model.input`) and the output file you should get when you run the model. The model describes the advection of two chemical substances. The first substance c_1 is emitted as a pollutant by a number sources. However, in this case this substance reacts with the oxygen in the air to form a more toxic substance c_2. The model implements the following equations:

$$\frac{\partial c_1}{\partial t} + u\frac{\partial c_1}{\partial x} = -1/Tc_1 \quad (7)$$

$$\frac{\partial c_2}{\partial t} + u\frac{\partial c_2}{\partial x} = 1/Tc_1 \quad (8)$$

- Run the model from the command line (passing input file as argument), not using OpenDA.
  The model generates the output files: `reactive_pollution_model.output` and `reactive_pollution_model_output.m`. You can create a movie of the model results using the `plot_movie.py` script from the `original_model` directory. This allows you to study the behaviour of the model.

For this exercise, the Java-routines for reading and writing the input and output files are already programmed. However, it is not necessary to program this in Java. It is also possible to write your own conversion program (in any programming language) to convert the input and output files of your model to a format that OpenDA is able to handle.

When you are interested in the actual java code that parses the input and output files of this black box model. You can find it at
`$OPENDA/model_reactive_advection` in a source distribution of OpenDA.

A black box wrapper configuration usually consists of three xml files. For our pollution model these files are:

1. `polluteWrapper.xml`: This file specifies the actions to performed when the model has to be run, and the files and related reader and writers that can be used to let OpenDA interact with the model.
   This file consists of the parts:

   - `aliasDefinitions:` This is a list of strings that can be aliased in the other xml files. This helps to make the wrapperxml-file more generic. E.g. the alias definition %outputFile% can be used to refer to the output file of the model, without having to know the actual name of that output file.
     Note the special alias definition %instanceNumber%. This will be replaced internally at runtime with the member number of each created model instance.
   - `run:` the specification of what commands need to be executed when the model is run.

- `inputOutput:` the list of 'input/output objects', usually files, that are used to access the model, i.e. to adjust the model's input, and to retrieve the model's results. For each 'ioObject' one must specify:
  - the java class that handles the reading from and/or writing to the file
  - the identifier of the ioObject, so that the model configuration file can refer to it when specifying the model variables that can be accessed by OpenDA, the so called 'exchange items' (see below)
  - optionally, the arguments that are needed to initialize the ioObject, i.e. to open the file.

2. `polluteModel.xml`: This is the main specification of the (deterministic) model. It contains the following elements:
   - `wrapperConfig`: A reference to the wrapper config file mentioned above.
   - `aliasValues`: The actual values to be used for the aliases defined in the wrapper config file. For instance the %outputFile% alias is set to the value "reactive_pollution_model.output".
   - `timeInfoExchangeItems`: The name of the model variables (the 'exchange items') that can be accessed to modify the start and end time of the period to that the model should compute to propagate itself to the next analysis time.
   - `exchangeItems`: The model variables that are allowed to be accessed by OpenDA, for instance parameters, boundary conditions, and computed values at certain locations. Each variable exchange item consists of its id, the ioObject that contains the item, and the 'element name', the name of the exchange item in the ioObject.
3. `polluteStochModel.xml`: This is the specification of the stochastic model. It contains of two parts:
   - `modelConfig`: A reference to the deterministic model configuration file mentioned above `polluteModel.xml`.
   - `vectorSpecification`: The specification of the vectors that will be accessed by the OpenDA algorithm. These vectors are grouped in two parts:
     - The state that is manipulated by an OpenDA filtering algorithm, i.e. the state of the model combined with the noise model(s).
     - The so called predictions, i.e. the values on observation locations as computed by the model.

Start with a single OpenDA-run to understand where the model results appear for this configuration:

- Have a look at the files `polluteWrapper.xml`, `polluteModel.xml` and `polluteStochModel.xml`, and recognize the various items mentioned above. Start the OpenDA GUI from the `public/bin` directory and run the model by using the `Simulation.oda` configuration. Note that the actual model results are available in the directory where the black box wrapper has let the model perform its computation: `stochModel/output/work0`.

In this exercise, we will calibrate the value of the reaction-rate constant. The algorithm used in this example is the Dud (which stands for Doesn't Use Derivative).

- Have a look at the `Dud.oda` and the configuration files it refers to. Run it from the OpenDA GUI and have a look at the results. What could you do to improve the results?
- Figure out where to change the control parameters for the calibration procedure and play around with the settings to improve your results.

Calibration runs normally take longer than a few minutes. In that case, it becomes convenient to be able to restart from a previous run.

- Adapt the configuration in such a way that you are able to restart the Dud.oda from the result of a previous run.

5 Exercise 5: A black box model - Filtering

We start with some single and ensemble runs to understand where for our black box wrapper configuration the model results appear:

- Run the model within OpenDA by using the `SequentialSimulation.oda` configuration. Use the script `plot_movie_seq.py` to visualize the model results. Compare the results with those from the run you executed without using OpenDA.
- Run an ensemble forecast model by using the `SequentialEnsembleSimulation.oda` configuration. On which variable does the algorithm impose stochastic forcing?
  Have a look at the `stochModel/output` directory, and note that the black box wrapper created the required ensemble members by repeatedly copying the template directory `stochModel/input` to `stochModel/output/work<N>`.
- Compare the result between the mean of the ensemble and the results from `SequentialSimulation.oda`. Note the relatively large differences. Check if these differences are reduced by increasing the ensemble size for the sequential ensemble simulation to 20 and rerunning `SequentialEnsembleSimulation..oda` (this run may take a few minutes). You can use the script `plot_movie_enssim.py`.

Now let us have a look at the configuration for performing OpenDA's Ensemble Kalman Filtering on our black box model, using a twin experiment as an example. The model has been run with the 'real' values (time dependent)for the concentrations for substance 1 as disposed by factory 1 and factory 2. This 'truth' stored in the directory `truthmodel`, and the results of that run have been used to generate observation time series at the output locations. These time series have been copied to the `stochObserver` directory to serve as observations for the filtering run.

The filter run takes the original model as input, which actually is a perturbed version of the 'truth' model: the concentrations for substance 1 as disposed by factories have been flattened out to a constant value. The filter process should

modify these values in such a way that the results resemble the truth as much as possible.

To do this the filter modifies the concentration at factory 2, and uses the observations downstream of factory 2 to optimize the forecast.

- Note that the same black box configuration is used for the sequential run, the sequential ensemble run, and for the EnKF run. Identify the part of the `polluteStochModel.xml` configuration that is used only by the EnKF run, and not by the others.

- Execute the Ensemble Kalman Filtering run by using the `EnKF.oda` configuration.
  Check how good the run is performing, by analyzing to what extent the filter has adjusted the predictions towards the observation.
  Note that the model output files in `stochModel/output/work0` only contains a few time steps. Can you explain why?
  So to compare the observations with the predictions you have to use the result file produced by the EnKF algorithm which can be visualised using `plot_movie.py`.

Now let us extend the filtering process by incorporating also the concentration disposed by factory 1, and by including the observation locations downstream of factory 1.

- Make a copy of the involved config files, `EnKF.oda` and `polluteStochModel.xml` (you could call them `EnKF2.oda` and `polluteStochModel2.xml`.
  Adjust `EnKF2.oda` so that it refers to the right stochastic model config file and produces a matlab result file with a recognizable name, e.g. `enkf_results2.m`.

- Now adjust `polluteStochModel2.xml` in such a way that the filtering process is extended as described above.

- Run the filtering process by using the `EnKF2.oda` configuration, and compare the results with the previous version of the filtering process.

6 Exercise 6: Writing your own toy model

Before you start:
In order to be able to compile your model you need to have a (current) version installed on your computer of:

- The Java Development Kit (JDK). You can download this from `www.oracle.com`[2]

- Apache Ant, this is a command line tool we use for building your java code. You can download Ant from `ant.apache.org`.

[2] Java Runtime Environment (JRE), which is installed on most computers is not sufficient since this will allow you to run java programs but it does not include the java compiler `javac` that is needed to create you own (parts of) programs

In this exercise you will learn how to code your own model and use it in OpenDA. The directory `exercise_6` contains a template of the code for the 1-D advection model we will create in this exercise. The content of this directory is similar to the OpenDA directories you have seen in the previous exercises. The difference is that we will not use a model that is already part of the OpenDA distribution but instead our own model. The model code can be found in the directory `simple_advection_model`.

The model you will create is build as an extension of the OpenDA `simpleStochModelInstance`. This will simplify and reduce the amount of programming because a significant part of the implementation is already available. For more complex models you might need to implement all methods of the `IStochModelInstance` class.

In the directory `exercise_6/simple_advection_model/java/src/org/openda` you will find the two java source files `AdvectionStochModelFactory.java` and `AdvectionStochModelInstance.java`. The first file implements the ModelFactory class. The model factory is a class in OpenDA that is responsible for creating model instances (e.g. the members of an ensemble Kalman filter). The second file implements the model. This is the file you have to edit in this exercise.

- Consider the 1-dimensional advection model:

$$\frac{\partial c}{\partial t} = v\frac{\partial c}{\partial x} \tag{9}$$

  where c typically describes the density of the particle being studied and u is the velocity. On the left boundary c is specified as $c_b(t) = 1 + \frac{1}{2}sin(5\pi t)$. Discretize this model on the interval $x = [0..1]$ with velocity $v = 1$ using a 1st order upwind scheme on a grid of 51 points. The time step is chosen such that the courant number $\frac{v\Delta t}{\Delta x}$ is approximately 1.

- The deterministic model is extended into a stochastic model by adding a noise parameter ω on the left boundary. We use an AR(1) model to describe the noise.

- Code your model in `AdvectionStochModelInstance.java`. For inspiration, you will find in the same directory an implementation of the Lorenz model.

- You can compile your model by typing "ant build" in the directory `exercise_6/simple_advection_model`. This will create the file `bin/simple_advection_model.jar`.

- Run the model. You can use file `advectionSimulation.oda`. In order to be able to run your model, java must be able to find the file `simple_advection_model.jar`. To accomplish this, you can copy your advection model jar-file to the bin-directory of your OpenDA installation or add the full path of your jar-file (`simple_advection_model.jar`) to the java class path variable `CLASSPATH`.
  By default, the Windows scripts `oda_run_gui.bat` and `oda_run_batch.bat` use the JRE environment that is provided with OpenDA. If this JRE is incompatible with your JDK installation (Error: Unsupported major.minor

version 51.0), use `oda_run_batch.bat <inputfile> -jre "location of your JDK"` to overrule the default JRE.

- Use the script `plot_movie.m` to visualize the model results. You will see that the model suffers from numerical diffusion. You can solve this by using a second order upwind method but this is not necessary for this exercise.
- Create an ensemble of model simulations and study the model uncertainty in space and time.
- The provided observation file `observations.cvs` does not contain observations that correspond to the advection model. Create your own observation file for the locations $x = 0.2$, $x = 0.5$ and $x = 0.7$. Use the model to create the values. Run the model with noise on the left boundary. Optionally generate some additional noise and add it to the generated observations. You can use the script `generate_obs` to simplify the creation of the observations file.
- Using your generated observations, setup an ensemble Kalman filter run. Experiment with various numbers of ensembles, different settings of the AR(1) model.
- Experiment with different intervals between the available observations. What do you observe. Is this behavior different from the Lorenz model? Evaluate the uncertainty of the estimates.
- Experiment with only assimilating the data from one of the three locations and use the other locations as validation. What do you observe. Do all the locations have the same impact? Explain the behavior you observe.
- Use the same data as generated before and use the Kalman filter now with different values of the system noise covariance and measurement noise covariance. Explain the behaviour that you observe.

7 Exercise 7: Localization

In this exercise you will learn about localization techniques and how to use them in OpenDA. This exercise is inspired on the example model and experiments from "Impacts of localisation in the EnKF and EnOI: experiments with a small model", Peter R. Oke, Pavel Sakov and Stuart P. Corney, Ocean Dynamics (2007) 57: 32-45.

The model we use is a simple circular advection model

$$\frac{\partial a}{\partial t} + u\frac{\partial a}{\partial x} = 0 \tag{10}$$

where u=1 is the speed of advection, a is a model variable , t is time and x is a space ranging from 1 to 1000 with grid spacings of 1. The computational domain is periodic in x.

In this model there are two related variables a and b where b is initialised with a balance relationship:

$$b = 0.5 + 10\frac{da}{dx} \tag{11}$$

and propagated with an advection model similar to the one for a, i.e.:

$$\frac{\partial b}{\partial t} + u\frac{\partial b}{\partial x} = 0 \tag{12}$$

Since a and b are propagated with the same flow, the balance relationship will remain valid also for $t > 0$. The relation ship between a and b is motivated by the geostrophic balance relationship between pressure (a) and velocity (b) in oceanographic and atmospheric applications.

In this experiment we will only observe and assimilate a and investigate how both a as b are updated. The ensemble is carefully constructed in order to have the right statistics. The initial ensembles are generated off line and they will be read when the model is initialised in OpenDA.

- Investigate the sript `generate_ensemble.py` and figure out how the ensembles are generated.
- Run python script `generate_ensemble.py` to generate ensembles, observations and true state for a 25, 50 and 100 ensemble experiment.
- Run the experiment for 50 ensemble members (`enkf_50.oda`).
- The variables a, b can be compared to the true state using the python script `plot_results.py`.
- Run the experiment for 25 ensembles, copy the script `plot_results.py` to e.g. `plot_results_25.py` and adjust it in order to read the results from `enkf25_results.py`. (change 2nd line of the `plot_results.py` script. You will see that the 25 ensemble run is not able to improve the model.
- Create input to run a 100 ensemble experiment. Note: do not forget to change the name of the output file (section `resultWriter`) to avoid that your previous generated results are overwritten.
- Run an experiment with 25 ensembles with localization (`enkf_25_loc.oda`) and generate the plots.
- The results (for 25 ensembles) with localization should look better than the the experiment without localization.
- Investigate whether the relation between a and b is violated by the various experiments. You can use the script `check_balance.py`.
- Try changing the localization radius (initial value is 50) and see how the performance of the algorithms changes (both for results as balance between a and b). You can plot the localization weight functions for each observation location (`rho_0`, `rho_1`, `rho_2` and `rho_3`) as well.