OpenDA course and exercises

Nils van Velzen, Martin Verlaan, Stef Hummel

July 15, 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`.

For postprocessing with python we assume that the numpy and pyplot modules are loaded. If not, then you can do this with the commands:

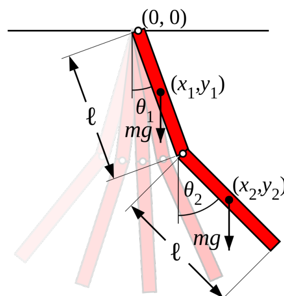
```
import numpy as np
import matplotlib.pyplot as plt
```

Listing 1: Python initialize

Please type (or copy-paste) onto the python prompt.

1 Exercise 1 part1: Getting started

A pendulum is a rigid body that can swing under the influence of gravity. It is attached at the top so it can roatate freely in a two-dimensional plane (x, y). We will assume a thin rectangular shape with the mass equally distributed. A double pendulum is a pendulum connected to the end of another pendulum. Contrary to the regular movement of a pendulum, the motion of a double-pendulum is very irregular when sufficient energy is put into the system.



The dynamics of a double-pendulum can be descibed with the following equations (This example was copied from https://en.wikipedia.org/wiki/Double_pendulum)

variables $\theta_1, \theta_2, p_{\theta_1}, p_{\theta_2}$:

$$\frac{d\theta_1}{dt} = \frac{6}{ml^2}\frac{2p_{\theta_1} - 3cos(\theta_1 - \theta_2)p_{\theta_2}}{16 - 9cos^2(\theta_1 - \theta_2)} \quad (1)$$

$$\frac{d\theta_2}{dt} = \frac{6}{ml^2}\frac{8p_{\theta_2} - 3cos(\theta_1 - \theta_2)p_{\theta_1}}{16 - 9cos^2(\theta_1 - \theta_2)} \quad (2)$$

$$\frac{dp_{\theta_1}}{dt} = -\frac{1}{2}ml^2\left(\frac{d\theta_1}{dt}\frac{d\theta_2}{dt}sin(\theta_1 - \theta_2) + 3\frac{g}{l}sin(\theta_1)\right) \quad (3)$$

$$\frac{dp_{\theta_1}}{dt} = -\frac{1}{2}ml^2\left(-\frac{d\theta_1}{dt}\frac{d\theta_2}{dt}sin(\theta_1 - \theta_2) + \frac{g}{l}sin(\theta_2)\right) \quad (4)$$

where the x, y-position of the middle of the two segments can be computed as:

$$x_1 = \frac{l}{2}sin(\theta_1) \quad (5)$$

$$y_1 = \frac{-l}{2}cos(\theta_1) \quad (6)$$

$$x_2 = l(sin(\theta_1) + \frac{1}{2}sin(\theta_2)) \quad (7)$$

$$y_2 = -l(cos(\theta_1) + \frac{1}{2}cos(\theta_2)) \quad (8)$$

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_pendulum_part1`. 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 double pendulum model. You can use the configuration file `simulation_unperturbed.oda`.

  For postprocessing in Matlab the results are written to `simulation_unperturbed_results.m`. Next start Matlab and load the results. We have added a routine `plot_movie` to create an intuitive representation of the data. Please type (or copy-paste):

```
[t,unperturbed,tobs,obs]= ...
load_results('simulation_unperturbed_results');
plot_movie(t,unperturbed)
```

  Listing 2: Matlab

  For postprocessing in Python the results are written to `simulation_unperturbed_results.py`. These results can be loaded with:

```
    import simulation_unperturbed_results as unperturbed
    # use reload(unperturbed) if unperturbed was loaded
  before
```

Listing 3: Python initialize

We have added a routine `plot_movie` to create an intuitive representation of the data.

```
    import pendulum as p #needed only once
    p.plot_movie(unperturbed.model_time,unperturbed.x)
```

Listing 4: Python

To create a time-series plot in Matlab type:

```
subplot(2,1,1);
plot(t,unperturbed(1,:),'b-');
ylabel('\theta_1');
subplot(2,1,2);
plot(t,unperturbed(2,:),'b-');
ylabel('\theta_2');
xlabel('time');
```

Listing 5: Matlab

To create a time-series plot in Python type:

```
    plt.subplot(2,1,1)
    plt.plot(unperturbed.model_time,unperturbed.x[:,0],'b'
  ) #python counts starting at 0
    plt.ylabel(r'$\theta_1$') # use raw string and latex
  for label
    plt.subplot(2,1,2)
    plt.plot(unperturbed.model_time,unperturbed.x[:,1],'b'
  )
    plt.ylabel(r'$\theta_2$')
    plt.show() #only needed if interactive plotting is off
  . Set with plt.ioff(), plt.ion()
```

Listing 6: Python

- Then you can start an alternative simulation with the double-pendulum model that starts with a slightly different initial condition using the configuration file `simulation_perturbed.oda`. The different initial conditions can be found in `modelDoublePendulumStochModel.xml` and `modelDoublePendulumStochModel_perturbed.xml`

- Visualize the unperturbed and perturbed results in a single plot. Make a movie and a time-series plot of θ_1 and θ_2 variables. Do you see the solutions diverging like the theory predicts?

```
    [tu,unperturbed,tobs1,obs1]=load_results('
simulation_unperturbed_results');
    [tp,perturbed,tobs2,obs2]=load_results('
simulation_perturbed_results');
    figure(1);clf;subplot(2,1,1);
    plot(tu,unperturbed(1,:),'b');
    hold on;
    plot(tp,perturbed(1,:),'g');
    hold off;
    legend('unperturbed','perturbed')
    subplot(2,1,2);
    plot(tu,unperturbed(2,:),'b');
    hold on;
    plot(tp,perturbed(2,:),'g');
    hold off;
```

Listing 7: Matlab

To create a movie with both results in python type:

```
  import simulation_unperturbed_results as unperturbed
  import simulation_perturbed_results as perturbed
  p.plot_movie(unperturbed.model_time,unperturbed.x,
perturbed.x)
```

Listing 8: Python initialize

To create a time-series plot with both results in Python type:

```
  plt.subplot(2,1,1)
  plt.plot(unperturbed.model_time,unperturbed.x[:,0],'b'
) #python counts starting at 0
  plt.plot(perturbed.model_time,perturbed.x[:,0],'g') #
python counts starting at 0
  plt.ylabel(r'$\theta_1$') # use raw string and latex
for label
  plt.subplot(2,1,2)
  plt.plot(unperturbed.model_time,unperturbed.x[:,1],'b'
)
  plt.plot(perturbed.model_time,perturbed.x[:,1],'g')
  plt.ylabel(r'$\theta_2$')
  plt.show()
```

Listing 9: Python

- Next we want to create an ensemble of model runs all with slightly different initial conditions. You can do this in a number of steps:
  - First create the input file `simulation_ensemble.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.
  - Create a configuration file for the Ensemble algorithm (e.g. named `algorithm/SequentialEnsembleSimulation.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/SequentialEnsembleSimulation.xml` in
    `simulation_ensemble.oda` and do not forget to give a diferent name to the output files.
  - Run the new configuration with OpenDA.
  - make a plot of the first and second variable of the five ensemble members in a single time-series plot

```
[t,ens]=load_ensemble('simulation_ensemble_results
');
ens_th1=reshape(ens(1,:,:),size(ens,2),size(ens,3)
);
ens_th2=reshape(ens(2,:,:),size(ens,2),size(ens,3)
);
clf; subplot(2,1,1);
plot(t(2:end),ens_th1);
ylabel('\theta_1');
subplot(2,1,2);
plot(t(2:end),ens_th2);
ylabel('\theta_2');
xlabel('time');
```

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
ens2=ens[:,1,:]
plt.subplot(2,1,1)
plt.plot(t[1:],ens1,'b')
plt.ylabel(r'$\theta_1$')
plt.subplot(2,1,2)
plt.plot(t[1:],ens2,'b')
plt.ylabel(r'$\theta_2$')
plt.show()
```

Listing 12: Python

– Observations of θ_1 and θ_2 are available as well. Make a plot of the observations together with the simulation results.

```
[t,unperturbed,tobs,obs]= ...
load_results('simulation_unperturbed_results');
    subplot(2,1,1);
    plot(t,unperturbed(1,:),'b-');
    hold on
    plot(tobs,obs(1,:),'k+');
    hold off
    ylabel('\theta_1');
    subplot(2,1,2);
    plot(t,unperturbed(2,:),'b-');
    hold on
    plot(tobs,obs(2,:),'k+');
    hold off
    ylabel('\theta_2');
    xlabel('time');
```

Listing 13: Matlab

```
  import simulation_unperturbed_results as
unperturbed
  plt.subplot(2,1,1)
  plt.plot(unperturbed.model_time,unperturbed.x
[:,0],'b') #python counts starting at 0
  plt.plot(unperturbed.analysis_time,unperturbed.obs
[:,0],'k+') #python counts starting at 0
  plt.ylabel(r'$\theta_1$') # use raw string and
latex for label
  plt.subplot(2,1,2)
  plt.plot(unperturbed.model_time,unperturbed.x
[:,1],'b')
```

```
    plt.plot(unperturbed.analysis_time,unperturbed.obs
[:,1],'k+')
    plt.ylabel(r'$\theta_2$')
    plt.show()
```

Listing 14: Python

We can see that although our simulation starts on the right track, it quickly diverges from the observations. The aim of the Ensemble Kalman filter or data-assimilation in general, is to keep the model on track.

2 Exercise 1 part 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

  Note that we are considering only uncertainty of the initial conditions here. In general, also uncertainty of the parameters or the model forcing, such as boundary conditions can be considered.

- Plot the ensemble mean of the first model variable and the observations. With some luck the solution should track the observations.
  For comparison we have also added the configurations for the 'truth' and a oda_run without data-assimilation called 'initial'.

```
    [tt,truth,tobs1,obs1]=load_results('
simulation_truth_results');
    [ti,initial,tobs2,obs2]=load_results('
simulation_initial_results');
    [te,enkf,tobs2,obs2]=load_results('
simulation_enkf_results');
    figure(1);clf;subplot(2,1,1);
```

```
        plot(tt,truth(1,:),'k');
        hold on;
        plot(ti,initial(1,:),'g');
        plot(te,enkf(1,1:2:end),'b');
        hold off;
        legend('truth','initial','enkf')
        subplot(2,1,2);
        plot(tt,truth(2,:),'k');
        hold on;
        plot(ti,initial(2,:),'g');
        plot(te,enkf(2,1:2:end),'b');
        hold off;
```

Listing 16: Matlab

```
        import simulation_truth_results as truth
        import simulation_initial_results as initial
        import simulation_enkf_results as enkf
        plt.subplot(2,1,1)
        plt.plot(initial.model_time,initial.x[:,0],'g')
        plt.plot(truth.model_time,truth.x[:,0],'k')
        plt.plot(enkf.analysis_time,enkf.x_f_central[:,0],'b
    ');
        plt.legend(('initial','truth','EnKF'))
        plt.ylabel(r'$\theta_1$')
        plt.subplot(2,1,2)
        plt.plot(initial.model_time,initial.x[:,1],'g')
        plt.plot(truth.model_time,truth.x[:,1],'k')
        plt.plot(enkf.analysis_time,enkf.x_f_central[:,1],'b
    ');
        plt.ylabel(r'$\theta_2$')
        plt.xlabel(r'$t$')
        plt.show()
```

Listing 17: Python initialize

- The Ensemble Kalman filter is uses a random number generator. In OpenDA we can control the initial value of the generator by adding a line like: `<initialSeed type="specify" seedValue="21" />` near the end of the main configuration file. Do you get the same results if you rerun with the same value of the initial seed? And what if you use a different value?
- 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 decrease it to 10 or 6?

3 Exercise 2: 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

$$\frac{\partial a}{\partial t} + u\frac{\partial a}{\partial x} = 0 \quad (9)$$

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:

$$b = 0.5 + 10\frac{da}{dx} \quad (10)$$

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

$$\frac{\partial b}{\partial t} + u\frac{\partial b}{\partial x} = 0 \quad (11)$$

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.

4 Exercise 3: A black box model - Filtering

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_5/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:

$$\frac{\partial c_1}{\partial t} + u\frac{\partial c_1}{\partial x} = -1/Tc_1 \tag{12}$$

$$\frac{\partial c_2}{\partial t} + u\frac{\partial c_2}{\partial x} = 1/Tc_1 \tag{13}$$

- 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`.

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.