OpenDA

The OpenDA Association

Apr 25, 2023

INTRODUCTION

1 Introduction to OpenDA 1

2 Introduction to data assimilation 3

3 OpenDA installation 5

4 Example configurations 7

5 Step-by-step application setup 11

6 OpenDA course 17

7 Netcdf data objects 19

8 Developing the Java source 23

CHAPTER

ONE

INTRODUCTION TO OPENDA

OpenDA is a generic environment for data-assimilation tasks like parameter calibration and measurement filtering. It provides a platform that allows an easy interchange of algorithms and models.

It is a modular framework, containing methods and tools that can be used for a wide range of applications. By offering the data-assimilation software as a separate component, the cost of applying data-assimilation methods in one's project is reduced. At the same time, it allows new developments in the field of data assimilation to quickly spread to all applications that might benefit from it. A short introduction to data assimilation can be found *here*.

OpenDA is configured using XML files (Extensible Markup Language files), in which the information about the data-assimilation components is specified. For instance, if you would like to use a different calibration algorithm or stochastic observer, or if you would like to couple your own model to OpenDA, you should provide all necessary settings, file names, variable names, etc. to OpenDA in XML input files. The format of the XML files is specified in XML schema files (`.xsd`) that are hosted on the OpenDA schemas website. The diagrams describing the format of the XML schemas can be found using an XML visualisation tool.

In general, the user needs to provide one main configuration file and several configuration files describing each data-assimilation component. The main configuration file contains references to the other components' configuration files. Usually, there are three main data-assimilation components: stochastic model, stochastic observer, and algorithm. In addition, another component may be specified to configure how OpenDA output will be stored.

In OpenDA, the following configuration files are used:

- Main configuration file (with XML schema `openDaApplication.xsd`): In the main configuration file, the OpenDA java class names, working directories and configuration file names of all the used data-assimilation components are specified.
- Stochastic observer: In this configuration file, the user specifies the observation data used in the application as well as the information about its uncertainty.
- Stochastic model: In this configuration file, the user specifies the model-related information.
- Algorithm: In this configuration file, the user specifies the input parameters required by the data-assimilation or parameter-calibration algorithm being used.

CHAPTER

TWO

INTRODUCTION TO DATA ASSIMILATION

Data assimilation is about the combination of two sources of information - computational models and observations - to utilize both of their strengths and compensate for their weaknesses.

Computational models are available nowadays for a wide range of applications: weather prediction, environmental management, oil exploration, traffic management and so on. They use knowledge of different aspects of reality, e.g. laws of physics, empirical relations, human behavior, etc., to construct a sequence of computational steps, by which simulations of different aspects of reality can be made.

The strengths of computational models are the ability to describe/forecast future situations (also to explore what-if scenarios), in a large amount of spatial and temporal detail. For instance, weather forecasts are run at ECMWF using a horizontal resolution of about 50 km for the entire earth and a time step of 12 minutes. This is achieved with the tremendous computing power of modern-day computers, and with carefully-designed numerical algorithms.

However, computations are worthless if the system is not initialized properly: “Garbage in, garbage out”. Furthermore, the “state” of a computational model may deviate from reality more and more while running, because of inaccuracies in the model, aspects that are not considered or not modeled well, inappropriate parameter settings and so on. Observations or measurements are generally considered to be more accurate than model results. They always concern the true state of the physical system under consideration. On the other hand, the number of observations is often limited in both space and time.

The idea of data assimilation is to combine a model and observations, and optimally use the information contained in them.

Some theory that might be helpful to understand the basics of data assimilation includes the distinction between offline and online methods, the statistical framework used, the notions of deterministic and stochastic models, noise models, the combination of values (weights are needed), data assimilation on top of an existing model, and the general structure of filtering methods.

An open-source text book considering the fundamentals of data assimilation can be found here.

CHAPTER

THREE

OPENDA INSTALLATION

On this page we explain how to install OpenDA on different platforms.

3.1 Windows installation

With the following instructions, the OpenDA software can be installed on a Windows machine:

- Download the OpenDA binaries. Note that the OpenDA binaries include a 64-bit Java 11 (or newer) installation to be used by the program.
- Extract the OpenDA distribution file to the desired location on your computer. Note: OpenDA does not work when it is installed on a location with a space in the path (like "`My Documents`").
- Open the OpenDA GUI by clicking the file `oda_run_gui.bat` in the folder `<path_to_openda_release>/bin`.
- Try to run an *example*.

3.2 Linux installation

With the following instructions, the OpenDA software can be installed on a Linux machine:

- Download the OpenDA binaries.
- Download a 64-bit Java 11 (or newer) installation via the package manager.
- Extract the OpenDA distribution file to the desired location on your computer. Note: OpenDA does not work when it is installed on a location with a space in the path (like "`My Documents`").
- Several system variables need to be set before OpenDA can be run.
  - The first variable that should be set is `$OPENDADIR`. This variable should point to the `bin` directory of your OpenDA installation:

    ```
    export OPENDADIR=<path_to_openda_release>/bin
    ```

    (make sure not adding a `/` after `bin`). By adding this line to the `~/.bashrc` file, this is done automatically each time a shell is launched.
  - Other local settings are configured using a script in the `bin` directory. There is a default local-settings script that might work out of the box for your system. You can use this script by typing `. $OPENDADIR/settings_local.sh linux` (mind the `.`). If that does not work, then the following steps should be executed:
    * Check your hostname, using the `hostname` command;

    * Copy the file `$OPENDADIR/settings_local_base.sh` to a new file named `settings_local_<hostname>.sh` in the same directory.
    * Then edit that file: enable the relevant lines and change the values of the environment variables.
    * Finally, execute this script by `. $OPENDADIR/settings_local_<hostname>.sh`.

- The OpenDA GUI can be opened by running `oda_run.sh -gui` in the folder `<path_to_openda_release>/bin`. Optionally the path to the `.oda` file can be supplied as an argument, which will open that OpenDA configuration (`oda_run.sh -gui <path_to_oda_file>`).
- Try to run an *example*.

3.3 Mac installation

With the following instructions, the OpenDA software can be installed on a Mac machine:

- Download the Linux OpenDA binaries (these binaries also work for a Mac).
- Download a 64-bit Java 11 (or newer) installation for Mac here.
- Extract the OpenDA distribution file to the desired location on your computer. Note: OpenDA does not work when it is installed on a location with a space in the path (like "`My Documents`").
- Several system variables need to be set before OpenDA can be run.
  - The first variable that should be set is `$OPENDADIR`. This variable should point to the `bin` directory of your OpenDA installation:

    ```
    export OPENDADIR=<path_to_openda_release>/bin
    ```

    (make sure not adding a `/` after `bin`). By adding this line to the `~/.bashrc` file, this is done automatically each time a shell is launched.
  - Other local settings are configured using a script in the `bin` directory. There is a default local-settings script that might work out of the box for your system. You can use this script by typing `. $OPENDADIR/settings_local.sh mac` (mind the `.`). If that does not work, then the following steps should be executed:
    * Check your hostname, using the `hostname` command;
    * Copy the file `$OPENDADIR/settings_local_base.sh` to a new file named `settings_local_<hostname>.sh` in the same directory.
    * Then edit that file: enable the relevant lines and change the values of the environment variables.
    * Finally, execute this script by `. $OPENDADIR/settings_local_<hostname>.sh`.
- The OpenDA GUI can be opened by running `oda_run.sh -gui` in the folder `<path_to_openda_release>/bin`. Optionally the path to the `.oda` file can be supplied as an argument, which will open that OpenDA configuration (`oda_run.sh -gui <path_to_oda_file>`).
- Try to run an *example*.

CHAPTER FOUR

EXAMPLE CONFIGURATIONS

On this page, we discuss some example configurations. They can be found in the directory `<path_to_openda_release>/examples`.

A first example that can easily be executed is found in `model_example_blackbox/blackbox_example_calibration/Dud.oda`. After opening this file in the *GUI*, you can run the simulation.

Note: for the examples `model_dflowfm_blackbox` and `model_delft3d`, it is necessary to connect to existing models.

Below we explain some more examples in detail.

4.1 Example 1: Oscillator-Dud application

In this example, we will learn how to use OpenDA for model calibration. To do so, we use an Oscillator model, which is one of the OpenDA internal/toy models. The algorithm used in this example is the Dud (which stands for **D** oesn't **U** se **D** erivative).

1. Check and examine the `core/simple_oscillator` directory. The main components in this directory are a main configuration file `Dud.oda` and three subdirectories that each contain a data-assimilation component: algorithm, model, and stochobserver. Each directory contains a configuration file for the respective component: `dudAlgorithm.xml`, `OscillatorStochModel.xml`, and `observations_oscillator_generated_for_calibration.csv`.
2. Run the OpenDA application with `Dud.oda` as the main configuration file in the mode that you prefer (with or without GUI). In GUI mode, you can get a real-time update of the execution by checking either Control, Output, or Cost Function tabs.
3. Check the results. Upon completion a new file `dud_results.m` is created. This file contains the results of OpenDA-Dud application, which are written in Matlab format. If you have no access to Matlab, then Octave can be used as an alternative.
4. Play around with the stopping criteria in `dudAlgorithm.xml` and see if the results are different from the previous ones. See the XML documentation for the description of each XML attribute.

4.2 Example 2: Oscillator-Simplex application

To edit the .xml files, the user is advised to use an XML validity editor, of which many can be found online (for free). In this example, we are going to use the same set of model and observation as in the previous one, but use a different algorithm for solving the parameter estimation problem. This will illustrate how easy it is in OpenDA to couple different algorithms to the existing model and observation.

1. Create a new algorithm configuration file: `simplexAlgorithm.xml`. Store it in the `algorithm` directory.
2. Edit the `simplexAlgorithm.xml`. See the XML documentation for the description of each XML attribute and an example of a simplex algorithm configuration file.
3. Create a new main configuration file `Oscillator-Simplex.oda`, by copying `Oscillator-Dud.oda`.
4. Edit `Oscillator-Simplex.oda`:
   - set the algorithm `className` to the one of simplex.
   - set another name for the `resultWriter configFile` to be different from the previous one to avoid overwriting the previous result.
5. Run the OpenDA application with `Oscillator-Simplex.oda` as input.
6. Check the results and compare them to the ones obtained using the Dud algorithm.

4.3 Example 3: Oscillator-Powell application

Follow the same procedure as in Example 2, but this time we use the Powell algorithm.

4.4 Example 4: Oscillator-GriddedFullSearch application

Follow the same procedure as in Example 2, but this time we use the Gridded Full Search algorithm.

4.5 Example 5: Oscillator-EnKF application

In the previous examples, we learned how to use OpenDA for model calibration or parameter estimation. In this example, we will learn how to use OpenDA for Kalman filtering. In this particular example, we use the Ensemble Kalman Filter (EnKF) algorithm. Follow the same procedure as in Example 2, but this time we use the EnKF algorithm. Check the results in `Enkf_results.m`. It contains several variables. For this tutorial, the following variables are of importance:

- `x_f{time}`: model state before data assimilation (forecast state)
- `x_a{time}`: model state after data assimilation (analysis state)
- `obs{time}`: observation data. Note that only one state variable is observed.
- `pred_f{time}`: forecast state variable which corresponds to the observed variable.
- `pred_a{time}`: analysis state variable which corresponds to the observed variable.

Notice that `pred_a` is much closer to the `obs` at each time than `pred_f`. This illustrates how data assimilation improves the model accuracy.

4.6 Example 6: Oscillator-EnSR application

Follow the same procedure in Example 6 for the Ensemble Square Root filter (EnSR).

4.7 Example 7: Lorenz-EnKF application

In the previous examples, we learned how to use different algorithms for the same set of model and observation. In this example, we will learn to couple another set of model and observation to the existing algorithms. We use the Lorenz model, which is another toy model available in OpenDA. Adjust the relevant XML attributes and elements in the main configuration files created in the previous examples to work with the Lorenz model and its available observation data `observations_lorenz_generated.csv`.

CHAPTER FIVE

STEP-BY-STEP APPLICATION SETUP

Setting up a data-assimilation framework for a model is a difficult task. Several things contribute to the complexity. The dynamical models are often complex software packages with many options. In addition, we add (real) observed data and blow up the number of computations and data by an order of 100. Another problem is related to our way of using the models in a data-assimilation framework. For sequential data-assimilation algorithms, such as the ensemble Kalman Filter (EnKF) or 3D-VAR, we often perform (short) model runs and update the parameters or state of the model between these short runs. However, the used simulation models are often not developed to be applied in this way.

Many users are struggling to get things to work because they want to do too much too soon. A recipe for failure is to attempt to set up your data-assimilation system for a real (big/huge) model with real data in one go. On this page, we describe some intermediate steps that can be taken to set up and test your application. There is no one-size-fits-all approach, but we try to present a recipe that you can use or adapt to your own needs. In between, there are useful tips and ideas.

For simplicity, we assume that the user uses a black-box coupling and wants to set up a data-assimilation system using a sequential data-assimilation algorithm e.g. a flavor of the ensemble Kalman filter.

5.1 Preparation

5.1.1 Start small

Setting up a data-assimilation system involves many steps and challenges. We advise not to focus on your final setup directly, which will involve real observations and probably a large simulation model. The last thing you will introduce is your real observations. You need their specifications e.g. location, quantity, quality, and sampling rate in an early stage since it is an important aspect of your system, but the measured values will not be used the first 80% of the time. To set up and test your framework it is best to use generated/synthetic data that you understand. We will explain this in more detail in the paragraph about setting up twin experiments below.

If possible, make various variations of your model. Start with a very, very simplified model that runs blisteringly fast and only incorporates the most basic features. When you have everything working for this small model, you move toward a more complex model. The number of steps you need to take to implement your full model depends on many aspects. Making these "extra" small steps is time well spent, and in our experience, you will save a lot of time in the end.

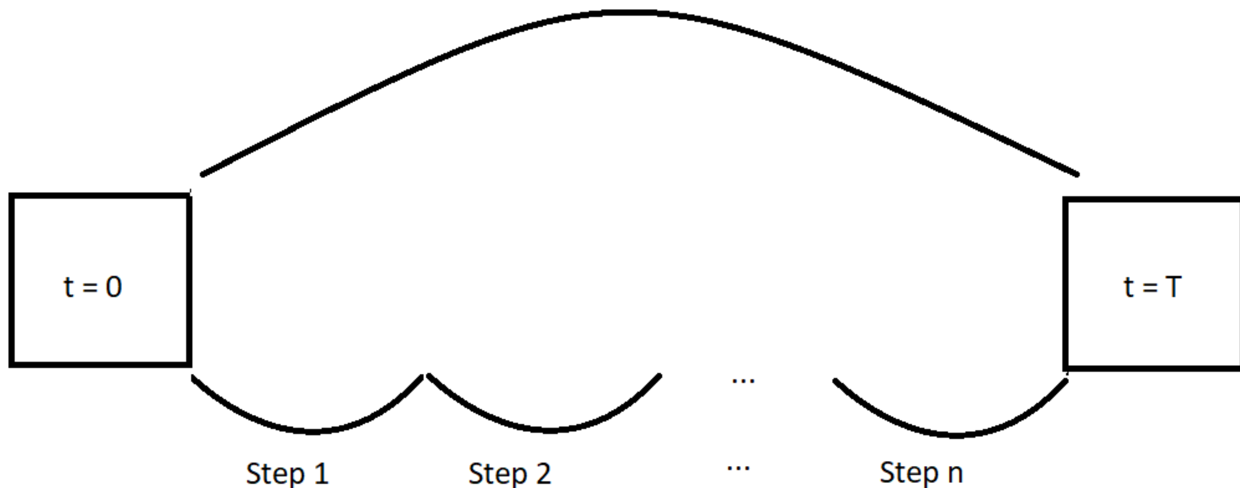
Create experiments with one group of observations at a time when you want to assimilate observations of various quantities and/or sources. You will learn a lot about the behavior of your model when assimilating these different types of observations, and it is much easier to identify which kind of observations might cause problems, like model instabilities.

5.1.2 Check the restart of a model

To use a programmed model as part of a sequential data-assimilation algorithm, it should have a proper restart functionality. This makes it possible to split up a long simulation run into several shorter ones. The model will write the internal state to one or more restart files at the end of each run. This will contain the model state x, but often some other information as well, e.g. the information on the integration step size, computed forcing, etc. The restart information will be read from disk at the start of the next run. There should be no differences in the result between the restarted simulations and the original simulation when the restart is implemented correctly.

To check whether the restart functionality of your model is working properly, run a simulation in one go (using the algorithm *Simulation*) and perform the same simulation with several restarts (using the algorithm *Sequential simulation*). It is always best to choose the same interval between the restarts as the assimilation interval you plan to use in your data-assimilation framework.

Note: In case your model software is already coupled to OpenDA it is still very important to check whether for your particular model setup the restart functionality works. Your model may be using features that impact the restart and have not been catered to in the original coupling to OpenDA.



Example of a proper restart functionality

Unfortunately, the restart functionality of models is often not perfect. When that is the case you have to look at how bad it is. Here is a list of issues we have seen in the past that might cause differences:

- Loss of precision: some precision can be lost in reading and writing values from the restart files (e.g. computations are in double precision but restart is in single precision). When we expect that the model updates of the data-assimilation algorithm are much larger than this loss of precision, it is only annoying (it makes testing/comparing/debugging more difficult) but no showstopper.
- Incomplete restart information: at some point in the model history some functionality has been added but the developers forgot to incorporate the relevant new (state) information in the restart file.
- Imperfect by design: sometimes, the developers never intended to have a perfect restart functionality, which means the results are not exactly the same as without the restart. Writing a correct restart functionality is often far from easy.

Some tips when you notice the restart is imperfect:

- Experiment with a simplified model. Switch features on and off to figure out where the differences are originating from.
- Does your model have automatic integration steps? Check the initial integration time steps for your restarted model. Can you run your model with constant integration time steps?

- How is the model forcing defined? Does the model interpolate your forcing input data? Changing the model time steps might fix your problems.
- Contact the developers of the code. With some luck, they are willing to help you.

In the end, you have to figure out whether the errors in the restart are acceptably small. When the deviation between the original run and a run with restarts is much smaller than the expected impact of your data assimilation you might be OK.

5.1.3 Uncertainty of your model

For the ensemble-based algorithms, you need to have an ensemble that statistically represents the uncertainty in your model prediction. There are various ways to set up your ensemble.

- When your model is dominated by chaotic behavior, e.g. for most ocean and atmospheric models, you can generate an initial ensemble by running the model for some time and taking various snapshots of the state. Another approach is to set up an ensemble of model states with some initial perturbation. Then run the ensemble long enough for the chaotic behavior to do its work and use that as the initial ensemble of your experiment.
- When the uncertainty is dominated by the forcing, e.g. coastal-sea, river, air-pollution, run-off and sewage models, you have to work on describing the uncertainty, including time and spatial correlations of these forcings.
- When the uncertainty is in the parameters of the model, e.g. groundwater and run-off models (and we are not planning to estimate them), you can carefully generate an ensemble of these parameters that represents their uncertainty. Then you set up your ensemble in such a way that each member has a different set of parameters. Be aware that this setup is not suited for all flavors of EnKF, since the model state after the update must in some sense correspond to the perturbed set of model parameters!

Combinations of the above are possible as well. It is a good investment of time to generate and explore your (initial) ensemble. Note that the filter can only improve your model based on the uncertainty (subspace) of your ensemble. When important sources are not captured by your ensemble, the filter will not be able to perform well.

Finally, your model may have time-dependent systematic errors. We often found it useful to add an artificial forcing to the model to describe these model errors.

We will explain *here* how these experiments can be carried out using OpenDA.

5.1.4 Twin experiments

In real-life applications, we use data assimilation to estimate the true state of the system. Unfortunately, we do not know the true state and that makes it difficult to test your data-assimilation system. You can set up a so-called *twin experiment* to overcome this problem and test your system in a controlled way. The observations in a twin experiment are generated by a model run with a known internal perturbed state or added noise. The perturbation should correspond to the specified uncertainty of your ensemble.

Note: Do not use the mean (or deterministic run), because that realization is special. The true state is known in the twin experiment and has the dynamics of your model. This makes it easy to investigate the performance of your data-assimilation framework. The *Sequential simulation* algorithm in OpenDA is a useful tool for creating your twin experiment.

5.2 Simulation algorithms

OpenDA implements several algorithms that can be used to gradually grow from a simulation model to a data-assimilation system.

5.2.1 Simulation algorithm

Running the algorithm `org.openda.algorithms.Simulation` is equivalent to running the model stand-alone. The only difference is that it runs from within OpenDA. It allows you to test whether the configuration is handled correctly and whether the output of the model can be processed by OpenDA.

5.2.2 Sequential simulation algorithm

The algorithm `org.openda.algorithms.kalmanFilter.SequentialSimulation` is again equivalent to running the model by itself. However, this time the model is stopped at each moment at which we have observations (or at predefined intervals). For each observed value, the corresponding value as predicted by the model is written to the output.

The purpose of this algorithm is twofold:

- Check whether the restart functionality of the model within the OpenDA framework is working correctly. This is done by comparing the results to a normal simulation.
- In addition, it is also used to create synthetic observations for a twin experiment. You set up observations with arbitrary values but with the location and time you are interested in. After you have run this algorithm, you can find the model predictions that you can use for your synthetic observations. Do not forget to add noise to the generated observations. The noise must be of the order of the measurement error when collecting 'real' data.

5.2.3 Sequential-ensemble simulation

The sequential-ensemble simulation algorithm (`org.openda.algorithms.kalmanFilter.SequentialEnsembleSimulation`) will propagate your model ensemble without any data assimilation. This algorithm helps you study the behavior of your ensemble. How is explicit noise propagated into the model? How is the initial ensemble propagated? At the same time, it is interesting to study the difference between the mean ensemble and your model run. Due to nonlinearities, your mean ensemble can behave significantly differently from your deterministic run.

5.3 Basic assimilation algorithms

5.3.1 Ensemble Kalman filtering

Next, it is time to start filtering. Therefore, ensemble Kalman filtering can be used (`org.openda.algorithms.kalmanFilter.EnkF`), but other algorithms, e.g. DEnKF or EnSR, are also possible. Start with a twin experiment so that you know that there are no artifacts in the observation data. Start small! First, assimilate a small number of observations and take the ones that may have a lot of impacts. Then start adding observations and see what happens. When you want to assimilate observations from various quantities or qualities, first investigate their impact as a group and only mix observations in the final steps.

5.3.2 Next steps

There are various methods and options that can help to improve the performance of your assimilation. To improve the performance you can try to use a pre-computed steady-state Kalman gain (`org.openda.algorithms.kalmanFilter.SteadyStateFilter`) or use parallel computing to propagate the ensemble in parallel. Spurious correlations can be catered to using localization techniques, which are available on most ensemble-based algorithms.

CHAPTER

SIX

OPENDA COURSE

A course is available to learn much more about OpenDA:

- `Course pdf with description and exercises`
- `Input files for the OpenDA course`

Some keywords:

- Configuration files
- Ensemble of model runs
- Observations
- Ensemble Kalman filtering
- Localization techniques
- Black-box model
- Writing wrappers
- Twin experiment
- Postprocessing in Python: time-series plots, movies

CHAPTER

SEVEN

NETCDF DATA OBJECTS

On this page, we describe the use of the netcdf data object. This data object is not specific to particular wrappers, but can be used for many different netcdf files. Here, the format of the variables should match one of the examples below when reading observations, model results or even updating states or boundary conditions. The netcdf data object is very useful since many different model software packages use netcdf files for storing and reading data.

In `org.openda.exchange.dataobjects.NetcdfDataObject`, the implementation of netcdf data objects can be found. It supports many different variable formats like scalar and gridded data. The most important ones will be described here.

7.1 Scalar data

The simplest form of scalar data has two dimensions, one for time and one for location. This will create exchange items for each location in each variable. The corresponding id is a combination of the variable name with the location name, which can be any string. The exchange item will contain the data for all the time steps. As an example, we create the exchange items

`24.waterlevel`, `25.waterlevel`, `26.waterlevel`, `27.waterlevel`, `28.waterlevel`, `29.waterlevel`

by using the following variables `waterlevel` and `data`:

```
float waterlevel(time=3, stations=6);
  :standard_name = "water_surface_height_above_reference_datum detection_minimum";
  :long_name = "waterlevel";
  :units = "m";
  :_FillValue = -9999.0f; // float
  :coordinates = "lat lon";
  :cell_methods = "time: maximum";

data:
  station_id = "26", "27", "28", "24", "29", "25"
```

with corresponding XML config:

```
<dataObject className="org.openda.exchange.dataobjects.NetcdfDataObject">
    <file>scalar.nc</file>
    <id>dataObjectId</id>
    <arg>true</arg>
    <arg>false</arg>
</dataObject>
```

Note:

- The options for `standard_name` are defined here. The `long_name` can be chosen.
- *Below*, an example of 3D scalar data is given.

7.2 Gridded data

Gridded data variables are supported with three dimensions containing time, x and y: in this case 1 exchange item will be created for the whole variable with the variable id as a name.

Example of a gridded variable:

```
double p(time=49, Y=16, X=27);
  :coordinates = "Time Y X";
  :long_name = "Atmospheric Pressure";
  :standard_name = "air_pressure";
  :units = "Pa";
```

with corresponding XML config (almost the same as for simple scalar, except for the filename):

```
<dataObject className="org.openda.exchange.dataobjects.NetcdfDataObject">
    <file>grid.nc</file>
    <id>dataObjectId</id>
    <arg>true</arg>
    <arg>false</arg>
</dataObject>
```

7.3 Extra arguments

In this section, we describe some extra arguments that can be used in the netcdf data object.

7.3.1 Boolean: Lazy reading and writing

The first two possible extra arguments are boolean values for lazy reading and lazy writing. When lazy reading is set to true, the data will only be read from the netcdf file when the data is needed instead of when initializing the data object. When lazy writing is set to true, the data will be written when the data object is closed instead of directly when the data is being changed by OpenDA. Depending on how much data will be read and/or written, how many times this will happen and the available memory, users can choose which settings will be most suitable.

These boolean arguments always have to be specified as the first 2 `<arg>` elements (lazy reading, lazy writing, respectively) in the XML config for the `NetcdfDataObject`, see the example *above*.

7.3.2 Key-value pair: `requiredExchangeItemId`

To prevent a long list of arguments with a specific order, key-value pair arguments have been introduced. Any number of `requiredExchangeItemId=<ID>` arguments can be supplied which limits the exchange items being created to the ones supplied. This can save a lot of memory and performance for large netcdf files.

If we use the same variables `waterlevel` and `data` as *above*, then it is possible to limit the creation of exchange items by adding three extra arguments: `requiredExchangeItemId=24.waterlevel`, `requiredExchangeItemId=26.waterlevel`, and `requiredExchangeItemId=27.waterlevel`. This way only the specified exchange items will be created.

The corresponding XML config looks as follows:

```
<dataObject className="org.openda.exchange.dataobjects.NetcdfDataObject">
    <file>scalar.nc</file>
    <id>dataObjectId</id>
    <arg>true</arg>
    <arg>false</arg>
    <arg>requiredExchangeItemId=24.waterlevel</arg>
    <arg>requiredExchangeItemId=26.waterlevel</arg>
    <arg>requiredExchangeItemId=27.waterlevel</arg>
</dataObject>
```

7.3.3 Key-value pair: `layerDimensionName`

For scalar data that have multiple layers, a `layerDimensionName=<name>` needs to be provided. The size of that dimension will determine the amount of layer-specific exchange items that will be created. When this argument is provided then the id of the exchange items will be constructed as follows: `<variableName>.<locationName>.layer<layerIndex>`

As an example, we take the next variables `temperature` and `data`:

```
double temperature(time=49, stations=3, laydim=20);
  :coordinates = "station_x_coordinate station_y_coordinate station_name zcoordinate_c";
  :units = "degC";
  :geometry = "station_geom";
  :_FillValue = -999.0; // double
  :standard_name = "sea_water_temperature";

data:
  station_id = "station01", "station02", "station03"
```

This will result in 60 exchange items with ids like:`station0i.temperature.layerj`, where i=1,2,3, j=0,…,19.

The corresponding XML config looks as follows:

```
<dataObject className="org.openda.exchange.dataobjects.NetcdfDataObject">
  <file>scalarLayers.nc</file>
  <id>dataObjectId</id>
  <arg>true</arg>
  <arg>false</arg>
  <arg>layerDimensionName=laydim</arg>
</dataObject>
```

7.3.4 Key-value pair: `allowTimeIndependentItems`

The argument `allowTimeIndependentItems=true/false` determines whether time-independent exchange items should be created. Time-independent exchange items can be created for variables that do not depend on a time dimension. The default is false.

CHAPTER

EIGHT

DEVELOPING THE JAVA SOURCE

On this page, we explain how developments in the Java source code can be made.

The OpenDA source code can be cloned from the OpenDA GitHub page.

The OpenDA software consists of four main modules. The first module is the `core` module, which contains the core of the OpenDA software. The three other modules are named after the conceptual components of data assimilation: `models`, `observers`, and `algorithms`. Each of these modules contains all programs and files related to the respective data-assimilation component. The `core` module contains programs which interface the other three modules. Modules for larger models with concrete applications are stored separately (`model_*`).

In addition to the source code, a 64-bit Java Development Kit (at least version 11) should be installed. Linux users can easily download JDK using the package manager, Windows users can download the software here.

8.1 Developing the Java source using IntelliJ IDEA

This page contains information about developing the OpenDA Java source code using IntelliJ IDEA. A free community version can be downloaded from the JetBrains website. By opening the project file`openda.ipr` in the main folder `<path_to_openda_source>`, most of the settings for this project will be set correctly.

In order to use the Java Development Kit in IntelliJ, you can refer to it via `File -> Project Structure -> Platform Settings -> SDKs`.

To build the source code into usable binaries, Ant is used which is already present in recent versions of IntelliJ. The Ant sidebar can be opened by heading to `View -> Tool Windows -> Ant`.

By selecting the `<path_to_openda_source>/build.xml` in the Ant plugin, a list of build targets can be run. Common practice is to run the `build-x64` target in this list by selecting it and choosing `Run Target`. By running it, the OpenDA binaries will be generated in `bin`. These binaries can be used to run OpenDA by non-developer users outside of IntelliJ.

If one of the XML schema files (`.xsd`) has changed, then it is necessary to run the Castor framework to convert XML files to Java source code. More information about these configuration files can be found in the *introduction to OpenDA*. Castor is run by selecting the corresponding `build_castor.xml` target in the Ant plugin (see, for example, `core/build_castor.xml`), and then executing it by choosing `Run Target`. After this step, the standard build target should again be run.

In each module folder (`core`, `models`, `model_*`, `observers`, and `algorithms`), unit tests are available to determine whether that particular part of the software is fit for use. The unit tests can be found in the folder `<path_to_module>/java/test/` and run by selecting `Run all tests` after clicking the right-mouse button.

8.2 Developing the Java source without an IDE

On this page, an explanation is given about the development of the Java source without using an IDE.

To build the OpenDA software, a command-line tool called Ant is used. Ant is similar to `make`, but written in Java, such that it is portable between different platforms. Linux users can easily install Ant using the package manager. Windows users can download the software here. Before installing Ant, make sure that the Java Development Kit is installed.

When Ant is installed, the complete source code can be built by simply typing `./ci_build.sh` (Linux) or `ant build` in the main `<path_to_openda_source>` (they do the same). Individual components (`core`, `models`, `model_*`, `observers`, or `algorithms`) can be compiled using `ant build` in the respective module directory.

If one of the XML schema files (`.xsd`) has changed, then it is necessary to run the Castor framework to convert XML files to Java source code. More information about these configuration files can be found in the *introduction to OpenDA*. Castor is run by executing `./ci_build_castor.sh` (Linux), or by visiting all directories containing a `build_castor.xml` file and executing `ant -f build_castor.xml`. After this step, the standard Ant build should again be performed.

To remove the files generated by a build, you can use `ant clean` on the command line. From within a module directory, this command will remove the `bin`, `build` and `javadoc` directories, and the `MANIFEST.MF` file (this file contains meta information about the used JDK version etc.). It does not affect other modules or the folder `bin` in the OpenDA main directory. Executing the command line `ant clean` from the OpenDA main directory will delete the folders `bin`, `doc`, and `xmlSchemas` in the main directory, as well as remove all modules' generated files. Note: to be able to delete files, they cannot be in use (obviously), so close them first.

In each module folder (`core`, `models`, `model_*`, `observers`, and `algorithms`), unit tests are available to determine whether that particular part of the software is fit for use. The unit tests can be found in the folder `<path_to_module>/java/test/`. The unit tests are run by executing an `ant test` script in the main OpenDA directory (see the `build.xml` for different versions, the original `ant test` should work). Separate tests are better run using IntelliJ.

genindex

modindex

search