OpenDA User Documentation

Date	:	August 31, 2015
Contact	:	**info@openda.org**

Contents

Chapter 1

Getting started with OpenDA

Contributed by:	
Last update:	00-0000

1.1 Introduction

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.

We assume that the reader is familiar with computational modeling, and with the principal aspects of data-assimilation: the distinction between off-line and online methods, the statistical framework used, the notions of deterministic and stochastic models, and the general structure of filtering methods.

1.2 Installation

The first thing that needs to be taken care of is the installation of OpenDA. Please read the appropriate installation page to see how that is done and whether it's done properly. We offer installation instructions for Linux, Mac and Windows.

1.2.1 OpenDA installation for Linux users

Note about csh

At the moment, scripts for csh and related shells are not included with the OpenDA distribution. It is possible to use OpenDA in conjunction with for instance tcsh, but you will have to convert the scripts yourself.

Note about GNU C Library

All native components of OpenDA are compiled for both 32-bit and 64-bit versions of Linux. For 32-bit systems a GLIBC version 2.4 or higher is needed, for the 64-bit version 2.7 or higher. If you have an older version of GLIBC, you need to compile the libraries yourself. It is important to remove all .so files in the lib directory before building.

Step-by-step installation

- Ensure that Java version 1.6 or higher is installed on your computer. You can check which version is installed through the java -version command. We have tested our software with the jre from SUN. The easiest way to install Java is often with the package manager that comes with your distribution (e.g. APT, yum for Red Hat-, dpkg for Debian- or YaST for SUSE-related distributions).
- Download OpenDA from www.openda.org. There is a script available to perform this download: openda_checkout_sf.sh
- Extract the OpenDA distribution file to the desired location on your computer. On some Linux systems unzip/ark is not installed by default, in that case try the package manager again.
- A number of 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. For example: export OPENDADIR=myhome/openda/bin

  The other variables are set by the script settings_local.sh in the directory $OPENDADIR. This script will try to call a local script with machine specific settings $OPENDADIR/settings_local_<hostname>.sh. You must create this script yourself.

  Copy the settings_local_base.sh file to a new file named settings_local_<hostname>.sh in your $OPENDADIR (unless that file already exists). You can check your hostname using the hostname command. Then edit that file: enable the relevant lines and change the values of the environment variables.

  For Linux there is a default local settings script that might work out of the box for your system. You can use this script by . $OPENDADIR/settings_local.sh linux.
- Most convenient is to have the variables set automatically. Add the following two lines to the .bashrc file in your home directory:

  export OPENDADIR=<bindir>, with <bindir> the location of the bin directory of your OpenDA installation.

  . $OPENDADIR/setup_openda.sh

  Note that the '.' is significant in the latter of these lines.

1.2.2 OpenDA installation for Mac users

Step-by-step installation

- Ensure that Java version 1.6 or higher is installed on your computer. You can check which version is installed through the java -version command.
- Download the OpenDA distribution from www.openda.org.
- Extract the OpenDA files to the desired location on your computer.
- A number of 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. For example:

  export OPENDADIR= /myhome/openda/bin.

  The other variables are set by the settings_local.sh script in the $OPENDADIR directory. This script will attempt to call the local script with machine specific settings $OPENDADIR/settings_local_<hostname>.sh. You must create this script yourself.

  Copy the settings_local_mac.sh file to a new file named settings_local_<hostname>.sh in your $OPENDADIR directory (unless that file already exists). You can check your hostname using the hostname command. Then edit that file: enable the relevant lines and change the values of the environment variables.

  The default local settings script might work out of the box for your system. You can use this script by typing:

  . $OPENDADIR/settings_local.sh mac.
- Most convenient is to have the variables set automatically. Add the following two lines to the .bashrc file in your home directory:

  export OPENDADIR=<bindir>, with <bindir> the location of the bin directory of your OpenDA installation.

  . $OPENDADIR/setup_openda.sh

  Note that the '.' is significant in the latter of these lines.

1.2.3 OpenDA installation for Windows users

Note about 64-bit systems

Windows XP is a 32-bit operating system, but some versions of Windows Server, Windows Vista and Windows 7 are 64-bit operating systems and can install the 64-bit version of Java. All native components of OpenDA are currently compiled for 32-bit versions of Windows only. If you need these native components on a 64-bit system, you have two options:

- Install a 32-bit version of Java.
- Compile the native libraries on your machine, see 2.8.3.

Step-by-step installation

- Ensure that Java version 1.6 or higher is installed on your computer, which you can check by typing java -version on the command-line. In case your Java version is too old, you can download the latest version from www.java.com/download
- Download OpenDA from www.openda.org
- Extract the OpenDA files 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").
- It is no longer needed to edit the start-up bat scripts, as in previous versions of OpenDA.
- If you use the command line (cmd.exe), then it is probably convenient to add the bin directory within the OpenDA directory to the PATH environment variable.

1.3 Using OpenDA

The next step will be learning how to use OpenDA. At first, a brief introduction to data assimilation is presented. The next step is an introduction to OpenDA itself. After these introductions, you should be ready to start OpenDA, either from the command-line or the graphical user interface (GUI). Some examples are presented to see how it works.

OpenDA is configured using XML files. 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 schemas (.xsd) files that are hosted on the OpenDA website. The diagrams describing the format of the XML schemas can found there as well.

1.3.1 Introduction to data assimilation

Data assimilation is about the combination of two sources of information - computational models and observations - in order 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., in order 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.

- off-line versus on-line;
- combine values: weights needed;
- statistical framework, std.error;
- deterministic versus stochastic models;
- noise model;
- data-assimilation on top of model.

1.3.2 Introduction to OpenDA

OpenDA is a generic environment for parameter calibration and measurement filtering. It provides a platform, where the interchange of algorithms as well as models can be done easily.

To use OpenDA, the user needs to prepare some configuration files, in which the information about the data assimilation components is specified. In general, 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. The OpenDA software makes use of the XML (Extensible Markup Language) format for the configuration files.

A number of configuration files are required to provide all necessary information about the OpenDA application. In general, the user needs to provide one main configuration file and a number of configuration files describing each data assimilation component. The main configuration file contains references to the other components configuration files.

The explanation of each configuration file is given below.

- Main configuration file (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.
- Algorithm
  In this configuration file, the user specifies the input parameters required by the data assimilation or parameter calibration algorithm being used.
- Stochastic Model
  In this configuration file, the user specifies the model related information.

OpenDA defines certain interfaces, which standardize how different components of OpenDA communicate with each other. For the model component, OpenDA defines two levels of interface: the model instance interface and the stochastic model instance interface. The model instance interface defines functionality which a (deterministic) model should implement. On the other hand, the stochastic model instance interface defines the stochastic extension of the deterministic model.

In order for a model to work within OpenDA, the model should be extended by implementing these interfaces. This is usually called wrapping the model. There are at least five ways one can wrap a model. The first way is to write the model code from scratch in Java and deliberately design the code to match OpenDA's requirements. The distribution of OpenDA contains several of such models. Those are small (toy) models, which are developed to test and illustrate various applications of OpenDA. The second way is to combine native model code with a wrapping Java extension. In this way, we keep the computation core of the model in its original code while extending it with a OpenDA wrapper. The third way is to write a model in Java, which implements only the model instance interface, and to use the existing Stochastic Black Box model utilities for its stochastic extension. The Black Box model utilities are various functions in OpenDA for implementing the required interfaces, which are generic and independent from the actual model. Making use of these utilities reduces the work one has to do to wrap a model. The fourth way is like the third one, but the computation core of the model uses the native code. The fifth way is the simplest method to wrap a model: a full black box model. In this way, one only needs to write several functions which read and/or write input and output files of the model. Once these methods are ready, one can simply use the Black Box model utilities to create a complete stochastic model extension required for a data assimilation application. While it is the easiest way to implement, an application based on the Black Box wrapper is the most computationally expensive. This is because the communication between the model and other OpenDA components is performed through writing and reading files.

The configuration files for the stochastic model depend on the type of model. For the toy models which are installed by default with OpenDA, there is only one model-configuration file needed. On the other hand, three configuration files are required for black-box models: the wrapper configuration, the model configuration, and the stochastic-model configuration. In the wrapper configuration file, the user needs to provide generic information about the model like aliases used to describe the model, the execution steps of the models relevant executable, and about input-output Java classes used by OpenDA to communicate with this model. In the model configuration, specific information of the particular model is given. The stochastic-model configuration file contains the information about vector specifications and may also contain information about the uncertainty of the model. For the DLL-based models, the configuration files needed depend on the choices made by the programmers of the OpenDA wrapper. In principle, they will require configuration files, where users can specify all the four data assimilation components mentioned above.

1.3.3 Starting OpenDA

Starting OpenDA on Windows

Run the oda_run_gui.bat batch file to open the OpenDA GUI. Optionally the path to the oda-file can be supplied as an argument, which will open that OpenDA configuration. The location of your system's Java Runtime Environment can be specified through -jre "path_to_jre".

Starting OpenDA on Linux/Mac

Run oda_run.sh gui to open the OpenDA GUI. Optionally the path to the oda-file can be supplied as an argument, which will open that OpenDA configuration.

1.3.4 OpenDA examples

You can find examples in the examples directory.

Chapter 2

Developers' corner

For people who want to (or have to) start from the OpenDA source code, the Developers' corner is added. The developers' corner is split in two parts: a Java section and a native section. The term native is used for parts of the source code that need to be compiled to a specific platform (Linux, Mac or Windows), formerly known as the Costa PSE.

2.1 Building Java sources

The OpenDA Java source code is located in the core/java directory of the source distribution, but when building everything, ant can be run from the OpenDA root directory (it will use the build.xml file located there).

OpenDA software consists of four different 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 contain 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.

Native libraries, written e.g. in C or Fortran are provided both as source and binaries with OpenDA. By default the build processes use the binaries. All binaries provided are 32-bit and therefore need a 32-bit version of Java. You can recompile the required native libraries as 64-bit if needed, for which you will need a 64-bit version of Java. For some blackbox-wrappers en small models you may not need any native libraries, so both 32-bit and 64-bit Java can be used.

2.2 Installation of Ant

To build OpenDA software, a command-line tool called Ant is used. Ant is similar to make, but written in Java, so that it is portable between different platforms. If Ant is not installed on your computer yet, you can download it from ant.apache.org/bindownload.cgi . For Linux users it is probably easier to use their package manager to install Ant. Before installing Ant, please check that a recent 32-bit version of Java is installed (which can be downloaded from www.java.com/download or in case of Linux, installed with the package manager).

2.3 Build commands

For the following description it is assumed that the OpenDA source is available in a directory named <path_to_openda>/public . Start by opening a command window (Windows) or a command shell (Linux) and change directory to the OpenDA main directory.

2.3.1 Compiling OpenDA

From the OpenDA main directory, you can compile all modules at once. Besides compilation there are several other options:

- ant (without any argument): to show help with list of possible ant arguments (the same as ant help).
- ant build: to compile OpenDA, make jar files and copy resources. This doesn't generate Javadoc documentation.
- ant doc: to build Javadoc for all modules and collect this documentation.
- ant clean: to remove the generated files.
- ant zip: to compile OpenDA, collect documentation and xml-schemas and create a set of zip files which contain the subversion revision number in the filename as well as in the name of the readme file. This makes it easy to wrap everything for exporting to a website or user. Exports with the same version numbers will extract to the same openda_<version> directory, but new versions can coexist.
- ant zip-tests: to create zip files for the various test cases. Each case is stored as separate zip file. Each file is named after the case followed by the version number.

2.3.2 Compiling individual components

From the module directory, you can compile a single module. Within a module directory the build file has somewhat different options:

- ant (without any argument): to show help with list of possible ant arguments.
- ant build: to compile the module, make jar files and copy resources. This doesn't generate Javadoc documentation.
- ant javadoc: to build Java doc.
- ant clean: to remove the generated files.
- ant make-standalone copy required external binaries to the module directory.

2.3.3 Compiling stand-alone modules

When providing others with a stand-alone module, you will have to provide other modules your stand-alone module depends upon (if any). The easiest way to achieve this, is by using ant make-standalone (after building all OpenDA modules) in the module directory. This will copy the files needed to the module directory.

2.4 Directory structure

Upon the execution of the command ant build or ant doc , the following folders are created in the main directory:

- bin: contains all binary files required for running OpenDA. This will be the content of OpenDA distribution file.
- doc: contains OpenDA documentation, including some examples.
- xmlSchemas: contains the XML Schema files for the OpenDA configuration files.

Each module directory has the following structure:

- bin: contains all binary files related to the respective module.
- build: contains the class files resulting from compiling the Java files of the respective module.
- javadoc: will contain Java documentation files when they are generated by executing ant javadoc on the command-line.

2.5 Removal of generated files

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. It does not affect other modules nor 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 removing all modules' generated files. Note: to be able to delete files, they cannot be in use (obviously), so close them first.

2.6 Native components

2.6.1 Brief introduction to OpenDA native components

Current practice in data assimilation software

Data assimilation techniques are widely used in various modeling areas like meteorology, oceanography and chemistry. Most implementations of data assimilation methods however are custom implementations specially designed for a particular model. This is probably a consequence of the lack of generic data assimilation software packages and tools. An advantage of these custom implementations is that they are in general very computationally efficient. But the use of custom implementations has a number of significant disadvantages:

- Costs - The development and implementation of these methods is very time consuming and therefore expensive.
- Incompatibility - It is very hard to reuse these data assimilation methods and tools for other models than they were originally developed for.

How OpenDA improves the situation

The OpenDA project tries to enhance the reuse of data assimilation software by offering a modular framework for data assimilation, containing methods and tools that can be easily applied for general applications. OpenDA is set up in order to be as computationally efficient as possible, without losing its generic properties. The aim is that applications developed with OpenDA have a comparable computational performance as custom implementations.

OpenDA offers support for both users and developers of data assimilation methods.

For users it allows models to be quickly connected to the OpenDA framework and hence to all the methods that are available in OpenDA. For developers, OpenDA offers efficient basic building blocks that save a lot of programming work and at the same time makes the new data assimilation software directly connectable to all OpenDA compliant models. How

OpenDA works OpenDA provides a generic framework for data assimilation. It is aimed both at model programmers that want to use data assimilation methods and at developers of data assimilation methods.

OpenDA for model programmers

For model programmers, OpenDA provides a rich set of data assimilation methods. To use them, you have to make your model OpenDA compliant. Once your model can interact with OpenDA, all the OpenDA methods are at your disposal.

What is OpenDA compliant?

Making your model OpenDA compliant involves implementing a number of routines that are going to be called from the data assimilation methods. The set of routines that you must implement is called the stochastic model interface. There are other interfaces as well, each one defining a specific entity called a component. You will read more about components in the next section.

How OpenDA calls your implementation of the interface routines

OpenDA connects your implementations of these methods to their standard names. These standard names are used in the implementation of the data assimilation methods. There are provisions for working with black-box models(i.e. models for which you do not have the source code), but this will not be discussed in this document.

Providing your observations

Likewise, your observations must be provided in a OpenDA compliant way. For the observations there is also a set of routines, called the stochastic observer interface. Usually, you will not implement the interface but convert your observations to a format that can be handled by the standard OpenDA implementation of the stochastic observer component.

OpenDA for developers of data assimilation methods

For developers of data assimilation methods, OpenDA offers a platform for quickly building data assimilation methods that can be used from a wide range of models. OpenDA provides various sets of routines that can be used as building blocks. Such a set is called a component. You will read more about components in the section Components.

Routines to interact with models

First of all, OpenDA specifies a set of routines to interact with the model to which the data assimilation must be applied. Each OpenDA compliant model implements these routines (otherwise it will not be OpenDA compliant). The specification of this set of routines is called the stochastic model interface.

Routines to interact with observations

A second set of routines that is essential for data assimilation methods comprises routines to interact with the observations. These include routines to retrieve values and routines to retrieve meta-information.

Basic building blocks

Finally, there are various sets of basic building blocks (e.g. for handling vectors, matrices and time). These interact seamlessly with the other OpenDA components to let you construct data assimilation methods with a minimum of coding.

2.7 Basic concepts of the OpenDA native components

2.7.1 The overall goals

The overall goal of OpenDA is to provide a toolbox with data assimilation capabilities which may be added easily to existing computational models:

- OpenDA should be applicable to a wide range of existing computational models.
  - complex, large-scale models should be supported as well as small scale test-models;
  - the models may be implemented using various programming languages, particularly Fortran, C/C++, Java;
  - the models may use parallel computing.
- It should be easy to get started, and one should quickly get initial results.
- One must be able to achieve good performance for large-scale models.

2.7.2 The overall philosophy

The basic design philosophy of OpenDA is illustrated in figure 2.1. The key elements in this picture are:

- a deterministic or stochastic model (red/white bricks);
- a collection of observations (red/white bricks);
- several data assimilation procedures (the grey bricks);
- the core of the OpenDA-system that connects the different building blocks (the yellow bricks).

The figure intends to illustrate a number of important points:

1. in OpenDA, data assimilation is implemented on top of existing model software;

2. the model and available observations need to be packaged in an appropriate way;
3. the different parts of the complete application are strongly separated from each other.

Other aspects, which are not yet illustrated in figure 2.1 are:

1. different "model builders" are provided for quickly packaging existing models and for combining separate models into larger ones;
2. techniques from object orientation are used for providing a basic framework that may be optimized for efficiency.

In OpenDA two components are defined: the OpenDA model component and the OpenDA stochastic observer component. There can be multiple instances of each of these components, each with their own set of data. The set of routines to manipulate the data is called the interface of the component. (Note: the terminology used in OpenDA with respect to object orientation is described in the paragraph about OO terminology).

OpenDA defines the interface of the OpenDA model component. This consists of the list of methods/routines that a model must provide. Examples are "perform a time-step", "deliver the model state", or "accept a modified state". This interface of the model component is visualized through the shape of the empty space in the yellow bricks in the figure above.

Usually, existing model code does not yet provide the required routines, and certainly not in the prescribed form. Therefore some additional code has to be written to convert between the existing code and the OpenDA model components interface. This is illustrated in the figure using red and white bricks: the white bricks stand for the original model code, whereas the red bricks concern the wrapping of the model in order to provide it in the required form.

OpenDA similarly prescribes the interface of the OpenDA observation component. This is visualized using a second empty space in the yellow bricks in the Lego figure. For using OpenDA for your model you must fill in this part, by wrapping the existing code for manipulation of your observations and providing it in the required form.

The data assimilation algorithms in the Lego figure cannot (yet) be seen as instances of another component. For now they are merely routines that implement different data assimilation techniques with the elements provided by OpenDA as the building blocks (models, observations, state vectors etc.). There is however a convergence to a fixed form, a more or less standardised argument list for data assimilation algorithms, such that eventually these may be turned into a well-defined component.

The basic design of OpenDA may seem disadvantageous at first, because it appears to require additional programming work in comparison to an approach where data assimilation is added to a computational model in an ad-hoc way. This is usually not the case. Most of the work in restructuring of the existing model code is needed in an ad-hoc approach too. This is because data assimilation itself puts requirements on the way in which the model equations are implemented in software routines: one must be able to repeat a time step, to adjust the model state, to add noise to forcings or the model state, and so on, which is often not true for computational models that are not implemented with data assimilation in mind.

The basic design does have a huge advantage over an ad-hoc approach for adding data assimilation to an existing program. It is that the different aspects of a data assimilation algorithm are clearly separated. The algorithmics of the precise Kalman filtering algorithm used are separated from the noise model, which in turn may be largely separated from the deterministic model and the processing of observations. This makes it easy to experiment with different choices for each of the parts: adjusting the noise model, adding observations, testing another data assimilation algorithm and so on. This is the major benefit of using the OpenDA approach.

2.7.3 Contents of the OpenDA toolbox

The yellow part in figure 2.1 provides the infrastructure needed for connecting generic model and observation components to generic data assimilation methods. We go into the contents of this part in order to gain a better insight in the structure and working of the OpenDA system.

The non-Java components of OpenDA are implemented using non-object oriented programming languages, particularly using Fortran and C. (One reason for this is to avoid difficulties when connecting model software that is written in Fortran and C, another motivation is the experience with Fortran and C of the original developers). However, OpenDA does use ideas from object orientation. In particular the following terminology is used:

- An ”object” is in OpenDA a variable in a computer program that cannot be manipulated directly, but only through the operations that are defined for it.
- A ”class” is the specification of a kind of objects. Objects are instantiations of the class to which they belong. The specification of a class describes both the data (properties, attributes) that objects of the class contain as well as the operations that may be performed.
- The term ”(software) component” is used in OpenDA to indicate classes which may involve a large amount of functionality. This concerns the ”OpenDA model component” and the ”OpenDA (stochastic) observer component”.
- The ”interface” of a class or a component is the set of routines that may be used to manipulate their instantiations.
- An ”OpenDA application” consists of a main program plus all the components, classes and other routines used. It may be packaged into a single executable, may be implemented using dynamic link libraries, or may be implemented in other forms as well.

Whereas the current OpenDA software and documentation often uses the word ”component”, Wikipedia (”Component based software engineering”, ”Class (computer science)”) suggests that ”class” would be more appropriate in most cases, with exceptions for the model and observer components. We suggest to adhere to the terminology introduced here.

OO Language	**OpenDA**
String s1, s2;	CTA_String s1, s2;
char cstr[100];	char cstr[100];
S1 = new String;	CTA_String_Create(&s1);
s2 = new String;	CTA_String_Create(&s2);
s1.Set(”hello ”);	CTA_String_Set(s1,”hello ”);
s2.Set(”world”);	CTA_String_Set(s2,”world”);
s1.Conc(s2);	CTA_String_Conc(s1, s2);
s1.Get(cstr);	CTA_String_Get(s1, cstr);
free(s1);	CTA_String_Free(s1);
free(s2);	CTA_String_Free(s2);

Illustration of the use of object orientation in the OpenDA native components: objects are instantiations of OpenDA-defined classes, are represented by object handles, and are manipulated using OpenDA-defined functions for the class.

As an example we consider the support for strings in OpenDA. A class CTA_String is provided for them. It is a simple container for character strings, which is primarily introduced to shield the difficulties of sharing text strings between Fortran and C. The operations provided for OpenDA strings are to create a new instance, set its value, concatenate strings, retrieve its value, and cleanup a string when it is not needed anymore. A piece of code using this class is presented in the table above.

Another basic class is the OpenDA time class, which provides a uniform way of handling time. The class registers a time span (interval) and optionally contains a time step attribute. It may be extended in a future version with various time scales and representations, time zones, etc..

A third basic building block is the OpenDA vector class, which contains a dimension (length), the data type of the vector elements (equal for all elements), and the values of the vector elements. An advantage of incorporating a vector class in OpenDA is that it provides a generic entity on which data assimilation algorithms can be based, without unnecessarily restricting oneself to the actual data type being used. Further, one can choose to provide different implementations (derived classes) of the vector class, for instance a distributed vector (for parallel computing, sparse vector (if a significant amount of zeros may occur, or using an optimized BLAS version.

An important class in the OpenDA toolbox is the OpenDA tree. It is a generic class for grouping and structuring of data. It may be compared to a ”struct” in C or derived type in Fortran. One notable difference is that a OpenDA tree is a dynamic entity: additional items may be added at run-time, with the names of the items provided as string arguments to the creation routines. Therefore a OpenDA tree may also be compared to a file system; nodes contain OpenDA trees (like directories), and leafs contain other OpenDA objects (like files).

A special kind (derived class) of OpenDA tree is the OpenDA tree-vector. It is a OpenDA tree that contains only OpenDA vectors as leaf elements. Of course it provides all the operations that a OpenDA tree class provides. Further it supports the operations that a OpenDA vector is able to perform. The OpenDA tree-vector is important for instance for describing a model state. In data assimilation one must be able to combine different model states much like vectors can be combined. However, representing the model state by a single vector is a severe restriction for many computational models. It is preferable to be able to distinguish different parts of the state, and sometimes needed to distinguish elements with different data types. Furthermore the hierarchical nature of the OpenDA tree-vector supports composition of larger models states out of smaller ones as described later on.

OpenDA primarily uses the XML-format for input/configuration-files. There are several facilities for quickly reading such input-files. OpenDA trees (and tree-vectors) may be written to and read from XML-files as well.

Interfaces are defined to the OpenDA classes for use in Fortran and C/C++. An interface for Java is defined separately in the OpenDA project. This allows data assimilation algorithms to be implemented in Java, and be used together with model components made in Fortran and C.

The OpenDA toolbox can be installed on various Linux and Unix-platforms using the well-known Automake facilities. For the Windows platform project files for Microsoft Visual Studio are provided.

2.7.4 Data assimilation methods and the OpenDA Application script

In the examples above OpenDA applications are created by writing a main program and calling OpenDA routines. In many cases it is more convenient to use the OpenDA Application script instead. This is a generic main program that may be configured to use your model, observations, and the requested data assimilation method.

The OpenDA Application script uses an XML-file to create new OpenDA applications. The XML-file defines the three main ingredients:

- The OpenDA model component to be used;
- the stochastic observer component to be used;
- the data assimilation method used.

The OpenDA Application script reads this configuration data, initializes the main model and observation components and then starts the data assimilation algorithm. The assimilation algorithm performs the actual work, and finally the Application script shuts down the computation.

Within the field of data assimilation a distinction between off-line and on-line assimilation methods can be made. Off-line data assimilation concerns model calibration, i.e. optimizing the parameters of a model such that the best fit with a set of observations is obtained.

In these methods the initial value of the model state may be considered as an input parameter of the model as well. This links the 3D-VAR and 4D-VAR variational approaches to the off-line methods listed above.

On-line or sequential data assimilation methods consist of a cycle of forecast and analysis steps, where the methods assimilates the data each time that observations become available. Optimal interpolation methods and Kalman filtering methods fall into this class.

2.7.5 The OpenDA stochastic model component

OpenDA uses a fixed form for a model component, in order to provide consistent terminology to both data assimilation methods and model implementers. The general form of a OpenDA model is $x(t+\Delta t) = M(x(t), p, u(t), w(t))$. Here x stands for the model state, t represents time, p is a vector of parameters, u concerns the forcings of the model, w is the noise/uncertainty, and M is the model operator. Simpler forms of models are possible, for instance a deterministic model without noise/uncertainty, a model without parameters or without forcings, and so on. Although this does not turn a model into an invalid OpenDA model, it may limit the data assimilation techniques that can be applied.

Data assimilation techniques will have to access the model state. This cannot be done directly. A model may use its own representation of the state. The interface of the OpenDA model component however requires that a model be able to provide a copy of the state in a OpenDA tree-vector, and that linear operations on a state-vector can be performed.

Linear operations on the model state are an important aspect of the interface of the model component because these operations are used frequently in data assimilation algorithms, and because the implementation may be dependent on the actual model that is implemented. For instance a model may require positivity of specific quantities that it computes (e.g. waterdepth in a flow model), such that blindly combining two state-vectors may give inappropriate results. Therefore a more careful combination recipe may be implemented by the model itself.

2.7.6 The use of native model builders

The idea of OpenDA is to make it simple to get started, but to provide full flexibility as well. This is implemented by providing default implementations where possible, but to also allow redefinition of the OpenDA components.

One place where this idea is given concrete form is in the concept of model builders. A model builder is more or less a template for creating new native model components. It fills in as many of the routines that must be provided, such that setting up a new model component is greatly simplified. Once one is up and running one can then tune the implementation to ones own ideas.

The sp-model builder

The SP model builder ("single processor") can be used to quickly create sequential (non-parallel) model components. This model builder defines choices for the storage and administration of the state vector, model parameters, changes to the forcings, and the noise parameters. With these choices made, a large portion of the methods that must be provided by a model component are provided by the model builder already, and only a few model specific methods have to be filled in.

The parallel model builder

The parallel model builder is meant for computational models that use parallel computing, and which probably must be started in an appropriate way. It is primarily meant for computational models that use MPI, which are based on multiple executables. Parallelization using multi-threading, especially using OpenMP, can often be used in OpenDA using a direct approach, for instance using the sp-builder.

Note that parallelization of data assimilation methods can often be achieved in different ways. A data assimilation algorithm often contains multiple model computations that may be performed in parallel, and each model computation itself may be parallelized too. The former approach to parallelization is called "mode-parallel". This name stems from the term "error modes" that may be used for different model instantiations in certain Kalman filtering algorithms. The latter approach using parallelization of the model computation itself is called "space-parallel", which indirectly refers to the domain decomposition approach. The two approaches may be combined as well, which gives a "mode and space-parallel" approach.

The parallel model builder is primarily concerned with space-parallelization. Mode-parallelization is already provided by the sp-builder ("using OpenDA, you get mode-parallelization for free"). It is provided by the parallel model builder as well, which yields the mode+space parallel approach.

The architecture used by the parallel model builder is to use an SPMD-approach ("single-program, multiple data"), i.e. to start the main OpenDA-application (executable) multiple times. Within the group of processes that is created in this way, the first one acts as the master process. This process performs the data assimilation algorithm. The other processes perform the model computations.

Computational models that use a master-worker approach for their parallelization may fit well into this approach. The master process of the original computational model may be integrated with the OpenDA master process. The subroutines that are required by the OpenDA model components interface may be filled in using the routines of the model's master process. This may be achieved well with the sp-builder.

The parallel model builder is mainly concerned with computational models that are parallelized using a worker-worker approach, e.g. using domain decomposition, where each computational process solves a different part of a global domain.

The routines that are specified in the OpenDA model components interface, like "perform a timestep", are implemented differently in the master and worker processes. In the master process, these routines consist of sending MPI messages to the worker processes, and waiting for the corresponding results. The results are joined together more or less the same as when different sub-models are combined into a composite model.

The worker processes read a configuration file from which they learn about their role in the global computation. Then they go into "worker-mode". This consists of an indefinite loop, waiting for MPI messages, and calling the appropriate model routines. The model routines are implemented just like for a sequential model. One notable difference is that these routines are called in all worker processes simultaneously, such that communication between the workers may be used.

The parallel model builder provides the infrastructure needed for setting up this scheme. It provides the model routines for the master process and the worker mode for the OpenDA main program. The model engineer just has to provide the model routines for the worker processes.

2.7.7 The OpenDA stochastic observer component

In OpenDA an observation is not just a value, but contains all the information available for use in a data assimilation method. This involves for instance information on the measurement error, which may be described by a probability density function. Other aspects of observations are the type of quantity that is observed, the unit, time, grid location, and so on.

The Stochastic Observer is the OpenDA component that holds an arbitrary number of observations. It may be instantiated multiple times.

A stochastic observer may be queried for a selection of the observations that it holds. For instance the observations within a given timespan may be requested, for a selected set of "stations", or that measure a specific quantity. Such a selection may be used to create a new stochastic observer object.

The stochastic observer further takes care of computing predicted values at observation locations. This concerns the observation relation that is used in data assimilation algorithms: the model state is interpolated and/or converted to the observation location and observed quantity. For this the implementation of a stochastic observer must know the structure of a model state vector, the meaning of its components, and the procedures available for spatial and/or temporal interpolation. For this a stochastic observer contains a substantial application dependent part.

OpenDA provides default implementations of the stochastic observer and observation descriptions. In this default implementation observations are stored in an SQLite3 database. The database contains two tables, "stations" and "data" for time-independent and time-dependent information respectively. This default implementation provides a basis for setting up ones own observation component. It should grow over time with features that are relevant to different computational models.

2.8 Building sources

2.8.1 Linux

This page describes how to build the OpenDA native source code on Linux computers. The source code is located in the core/native directory of the source distribution. On Linux, the native sources are compiled using the GNU Automake system.

Directory scripts of the source distribution contains a script *$openda_build_native.sh$* that may be useful to compile the native code. It was tested for Ubuntu 8.04 lts 32-bit only. In this script, you will recognize the steps described below.

Building step-by-step

1. The first step is starting the configure script, usually through ./configure (to ensure the script you are starting is the one in the current directory and not another one from the search path). This will detect the configuration of the computer being used and will warn when specific requirements are not met. When all requirements are met, make files will be generated. It is possible to alter the behaviour of the configure script by using command-line arguments. The most important ones are:

   - –help will list all options with some help text.
   - –prefix=PATH indicates the place the library should be copied to after make install.
   - –disable-mpi disables MPI.
   - –with-blas=PATH indicates the location of the BLAS library. By default, an unoptimized BLAS library is used.
   - –with-lapack=PATH indicates the location of the LAPACK library, an unoptimized LAPACK library is used.
   - –with-jdk=PATH indicates the location of the Java Development Kit (JDK) if it differs from the value of $JAVA_HOME.
   - –with-jikes=PATH indicates the location of the Jikes Java compiler in case that compiler is to be used. Default: no.

   Do not forget to scan the configure output for warnings. Those are often very informative.

2. The second step is using make to build (compile and link) the source files.

3. The final step is copying the resulting libraries (and executables) to the place specified using configure's –prefix= command-line argument. This step is activated by make install.

The Automake system also generated the other usual make options (like make clean). It is unlikely that you want to remove the libraries and executables you just built, but in case you want to, this is nice to know.

Note about OpenMPI

There is a known problem with OpenMPI versions 1.3 and 1.4 where an external dependency mca_base_param_reg_int cannot be found during run-time. This can be avoided by recompiling OpenMPI itself. Use command-line arguments –enable-shared –enable-static when running the config script.

2.8.2 Mac

This page describes how to build the OpenDA native source code on Mac computers. The source code is located in the core/native directory of the source distribution. On Mac computers, the native sources are compiled using the XCode development environment.

Preliminaries

1. Install XCode.
2. Install a GFortran compiler that is compatible with your Xcode installation.
3. Install OpenMPI (the same version as Xcode) with fortran support (–ensable-static –enable-dynamic). and insert the path in front of your %PATH% and %LD_LIBRARY_PATH% variables.
4. Install a Java Development Kit (JDK).

Building step-by-step

1. The first step is starting the configure script, usually through ./configure. This will detect the configuration of the computer being used and will warn when specific requirements are not met. When all requirements are met, make files will be generated. It is possible to alter the behaviour of the configure script by using command-line arguments. The most important ones are:
    - –help will list all options with some help text.
    - –prefix=PATH indicates the place the library should be copied to after make install.
    - –disable-mpi disables MPI.

- –with-netcdf=PATH indicates the location of the NetCDF library.
- –with-jdk=PATH indicates the location of the Java Development Kit (JDK) if it differs from the value of $JAVA_HOME.

Do not forget to scan the configure output for warnings. Those are often very informative.

2. Build (compile and link) the source files.

3. Copy the resulting libraries (and executables) to the place specified using configure's –prefix= command-line argument.

2.8.3 Windows

This page describes how to build the OpenDA native source code on Windows computers. The source code is located in the core/native directory of the source distribution. The Microsoft Visual Studio solution file is located at core/native/vs2010/OpenDANativeAll.sln .

Note about Microsoft Visual Studio and Intel Fortran

The solution and project files provided are for the following version of the development environment:

- vs2010 and Intel Fortran Composer 13 or higher

When you use a newer version of mentioned tools, it is fine to upgrade these files to your version. In case you are working from the repository, please do not check in these upgraded files.

Building step-by-step

1. Load the solution file core/native/vs2010/OpenDANativeAll.sln into Microsoft Visual Studio.

2. Select whether you want to perform a Release build or a Debug build.

3. Start the build process with Build All.

Note about the Intel Fortran library path

In some installations of Microsoft Visual Studio, the Intel Fortran library path is not added to the library path during the installation (and integration) of Intel Visual Fortran. This will lead to a link error about one or more missing libraries, usually ifconsol.lib,

If this is the case, solve it by either:

- Add the lib directory of the Intel Fortran installation to the global library path in Microsoft Visual Studio. This path can be found in Visual Studio menu path Tools/Options/Projects and Solutions/VC++ Directories/Library files.
- Add the lib directory of the Intel Fortran installation to the project's (probably libcta's) library path. This path can be found the project's right-click menu: path Properties/Configuration Properties/Linker/General/Additional Library Directories.

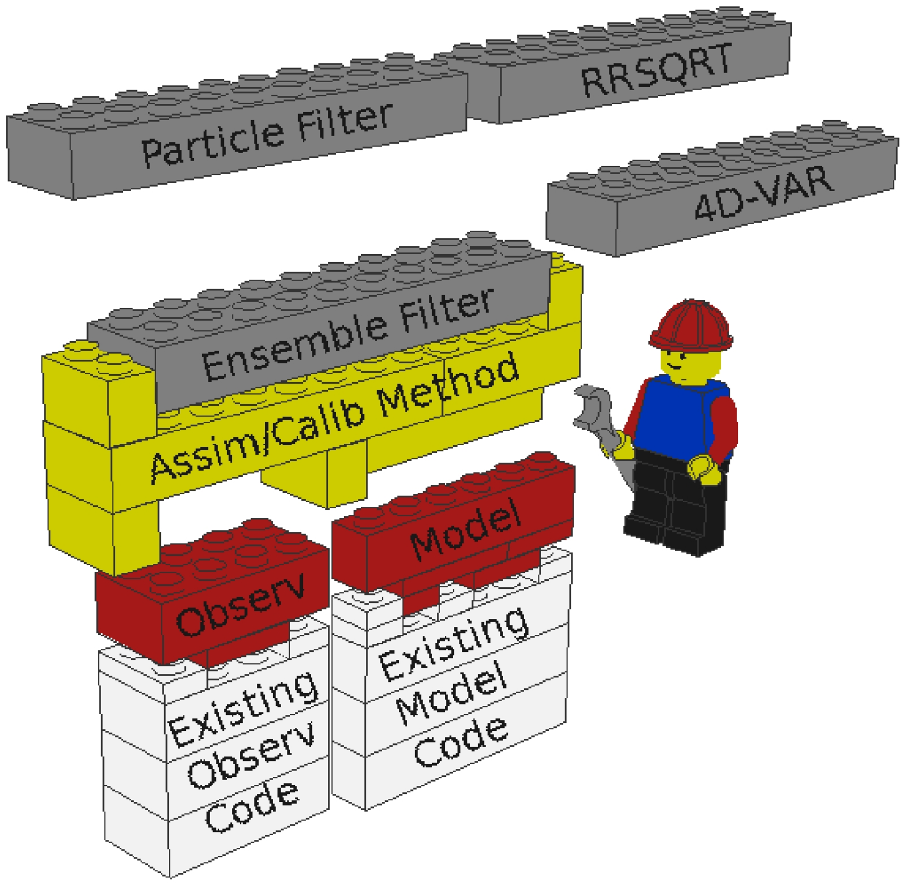

Figure 2.1: Basic design of the OpenDA system. Model and observation components (red/white bricks) are plugged into the core of the OpenDA environment (yellow bricks), and can then exploit the available data-assimilation methods (grey bricks).

Chapter 3

Toy models available in OpenDA

Origin:	CTA memo200802
Last update:	00-0000

A number of small toy models are available in OpenDA, meant for testing and teaching purposes. All models are represented as a set of differential equations where the solution is obtained numerically by using the Runge-Kutta method or Forward Euler.

3.1 Oscillator model

The oscillator model is a simple mass-spring model with friction. It has two describing variables, which are the location of the mass x and its velocity u. The two variables are related according to the following equations:

$$\frac{dx}{dt} = u \tag{3.1}$$

$$\frac{du}{dt} = -\omega^2 x - \frac{2}{T_d}u \tag{3.2}$$

where ω is the oscillation frequency, which depends on the mass and the spring constant, while T_d is the damping time.

3.2 Lorenz model

Edward Lorenz (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:

$$\frac{dx}{dt} = \sigma(y - x) \tag{3.3}$$

$$\frac{dy}{dt} = \rho x - y - xz \tag{3.4}$$

$$\frac{dz}{dt} = xy - \beta z \tag{3.5}$$

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.

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.

3.3 Lorenz96 model

The Lorenz96 model (Lorenz and Emanuel (1998)) is defined by the following equation

$$\frac{dx_i}{dt} = x_{i-1}(x_{i+1} - x_{i-2}) - x_i + F \tag{3.6}$$

where $i = 1, ..., N$, $N = 40$ and the boundary is cyclic, i.e. $x_{-1} = x_{N-1}$, $x_o = x_N$, and $x_{N+1} = x_1$, and $F = 8.0$. The first term of the right hand side simulates "advection", and this model can be regarded as the time evolution of an arbitrary one-dimensional quantity of a constant latitude circle; that is, the subscript i corresponds to longitude. This model also behaves chaotically in the case of external forcing $F = 8.0$.

3.4 Heat transfer model

The model represents a special case of heat propagation in an isotropic and homogeneous medium in the 2-dimensional space. The equation can be written as follows:

$$\frac{\partial T}{\partial t} = k(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}) \tag{3.7}$$

for $x \in [0, X]$ and $y \in [0, Y]$ where T is the temperature as a function of time and space and k is a material-specific quantity depending on the thermal conductivity, the density and the heat capacity. Here k is set to 1. Neumann and Dirichlet boundary conditions are used.

3.5 1-dimensional Advection model

In this study, a 1-dimensional advection model is also used and can be written as follows

$$\frac{\partial c}{\partial t} = u\frac{\partial c}{\partial x} \tag{3.8}$$

where c typically describes the density of the particle being studied and u is the velocity. On the left boundary c is specified as $c_b(t) = 1 + sin(\frac{2\pi}{10}t)$.

3.6 Stochastic Extension

The previous subsections describe the deterministic models available within OpenDA. Especially for the implementation of Kalman filtering, we need to extend the models into a stochastic environment. This is done, for the oscillation, Lorenz, and Lorenz96 models, by adding a white noise process to each variable. On the other hand, for the heat transfer and 1-d advection models, this is done by adding a colored noise process, represented by an AR(1) process, to the boundary condition. For the heat model the noise is also spatially correlated.

Chapter 4

Calibration methods available in OpenDA

Origin:	CTA memo200802
Last update:	00-0000

In the calibration or parameter estimation algorithms, the basic idea is to find the set of model parameters which minimizes the cost function measuring the distance between the observation and the model prediction. Two different cost functions are implemented. The first one is similar to equation (5.1), while in the second one we add the background component as follows

$$J(x_o) = (x_o - x^b)'(P^b)^{-1}(x_o - x^b) + \sum_{k=1}^{N}(y^o(k) - Hx(k))R^{-1}(y^o(k) - Hx(k))' \tag{4.1}$$

Where x^b and P^b are the background or initial estimate of x_o and its covariance respectively. This additional component ensures that the solution will not be too far from the initial guess.

The following calibration methods are implemented in OpenDA:

- Dud
- Sparse Dud
- Simplex
- Powell
- Gridded full search
- Shuffled Complex Evolution (SCE)
- Generalized Likelihood Uncertainty Estimation (GLUE)
- (L)BFGS
- Conjugate Gradient: Fletcher-Reeves, Polak-Ribiere, Steepest Descent

4.1 Parameter estimation with the Simplex method

The simplex method that is implemented in COSTA is the one due to Nelder and Mead (Nelder and Mead (1965)). It is a systematic procedure for generating and testing candidate vertex solutions to a minimization problem. It begins at an arbitrary corner of the solution set. At each iteration, the simplex method selects the variable that will produce the largest change towards the minimum solution. That variable replaces one of its compatriots that is most severely restricting it, thus moving the simplex to a different corner of the solution set and closer to the final solution.

A simplex is the geometrical figure consisting, in N dimensions, of $N + 1$ points and all their interconnecting line segments, polygonal faces, etc. In two dimensions, a simplex is a triangle. In three dimensions it is a tetrahedron. The simplex method must be started with $N + 1$ points of initial guess, defining the initial simplex. The simplex method now takes a series of steps. The first step is to move the vertex where the cost is largest through the opposite face of simplex to a lower point. This step is called the *reflect* step. When the cost of the new vertex is even smaller than all the remaining, the method expands the simplex even further, called the *expand* step. If none of these steps produce a better vertex, the method will contract the simplex in the same direction of the previous step and take a new point. This step is called the *contract* step. When the cost of the new point is still not better than the previous one, the method will take the last step called *shrink*. In this step all of the simplex points, except the one with the lowest cost, are 'shrinked' toward the best vertex.

As it basically only tries and compares the solution of several different sets of parameters, the method requires only function evaluations and not derivatives.

4.1.1 Using the Simplex method

For using the implemented simplex method the user needs to specify in the input file the *initial-guess* of the set of parameters to calibrate as well as the *initial-step* for creating the initial simplex. The initial-step consists of N entries, where N is the number of parameters to calibrate. An initial vertex is obtained by adding an element of the initial-step to the corresponding parameter in the initial-guess. Performing this one by one to all the parameters in the initial-guess, there are $N + 1$ vertices forming the initial simplex. Although it can be easily extended, at the moment the program only supports the model with a maximum of 10 parameters.

The stopping criteria for the iteration are the maximum number of iteration and the maximum cost difference between the worst and the best vertices, where worst vertex refers to the one with the biggest cost value and best vertex is the one with the lowest. When the maximum cost difference is very small we may expect that the (local) minimum of the cost function has been reached. The optimum solution is the vertex with the lowest cost. However, as output we display all the final vertices with their respective costs.

Since the model parameters are stored as COSTA state vector, most variable operations are performed in term of COSTA state vector. The operations are usually to compute new vertex. This can easily be carried out by using the combination of functions like cta_state_axpy and cta_state_scal, with the aid of functions like cta_state_duplicate and cta_state_copy for assigning values to working variables. These COSTA functions reduce significantly the lines of codes required to perform the operation.

An important function required to implement the simplex method is the sorting function. This function is used to sort the vertices with respect to their cost values in descending order. While the vertices are stored as COSTA state vectors, their cost values are stored as a Fortran array. The sorting is done by using an external routine, which works only with Fortran variables. The output of this routine are the sorted array of cost values and an integer array containing the index of the sorted array.

In the section above we learnt that the simplex method consists of several steps which are taken to find a new vertex with lower cost. When all the steps do not produce a better vertex the program will give a warning. For some cases this may indicate that the solution does not exist. This occurs for example with the Lorenz model. Since it is a chaotic model, small difference in the parameters yields very different cost values. This makes the Lorenz model not a very good model for calibration tests in the present setup. Perhaps shortening the interval over which the optimization is carried out can make the cost function better behaved.

4.1.2 The configuration of the Simplex method

- `simulation_span`: the overall timespan to run the model
- `output`: output specification
- `iteration`: Number of iterations and convergence tolerance
- `model<n>`: Model configuration for the initial model state at every starting vertex

4.1.3 XML-example

```
<parameter_calibration>
  <CTA_TIME id="simulation_span" start="0" stop="50.0" step=".1"/>
  <output>
      <filename>results_oscill.m</filename>
     <CTA_TIME id="times_mean" start="0" stop="50.0" step="1"/>
  </output>
  <iteration maxit="60" tol="0.001">    </iteration>
  <!-- 1st VERTEX:-->
  <model1>
     <modelbuild_sp>
```

```
        <xi:include href="models/functions_oscill.xml"/>
         <model>
            <parameters avg_t_damp="8.95" avg_omega="13.5"> </parameters>
         </model> -->
      </modelbuild_sp>
   </model1>
   <!-- 2nd VERTEX:-->
   <model2>
      <modelbuild_sp>
        <xi:include href="models/functions_oscill.xml"/>
         <model>
            <parameters avg_t_damp="6.0" avg_omega="15.9"> </parameters>
         </model> -->
      </modelbuild_sp>
   </model2>
   <!-- 3rd VERTEX:-->
   <model3>
      <modelbuild_sp>
        <xi:include href="models/functions_oscill.xml"/>
         <model>
            <parameters avg_t_damp="9.0" avg_omega="12.5"> </parameters>
         </model> -->
      </modelbuild_sp>
   </model3>
 </parameter_calibration>
```

4.2 Parameter estimation with the Conjugate Gradients method

The problem of minimization of multivariable function is usually solved by determining a *search direction* vector and solve it as a line minimization problem. If x is a vector containing the variables to be determined and h is the vector of search direction, at each iteration step the minimization problem of a function f is formulated as to find the step size λ that minimizes $f(x+\lambda h)$. At the next iteration, x is replaced by $x+\lambda h$ and a new search direction is determined. Different methods basically propose different ways of finding the search direction.

The conjugate gradient method is an algorithm for finding the nearest local minimum of a function which uses *conjugate directions* for going downhill. Two vectors u and v are said to be conjugate (with respect to a matrix A) if

$$u'Av = 0 \tag{4.2}$$

where in the minimization problem, A is typically the *Hessian* matrix of the cost function. In the conjugate gradient methods, the search direction is somehow constructed to be conjugate to the old gradient.

The two most important conjugate gradient methods are the *Fletcher-Reeves* and the *Polak-Ribierre* methods (TODO: Press et.al., 1989). These algorithms calculate the mutually conjugate directions of search with respect to the Hessian matrix of the cost function directly from the function and the gradient evaluations, but without the direct evaluation of the Hessian matrix. The new search direction h_{i+1} is determined by using

$$h_{i+1} = g_{i+1} + \gamma_i h_i \tag{4.3}$$

where h_i is the previous search direction, g_{i+1} is the negative of local gradient at iteration step $i + 1$, while γ_i is determined by using the following equations for *Fletcher-Reeves* and the *Polak-Ribierre* methods respectively:

$$\gamma_i = \frac{g_{i+1} \cdot g_{i+1}}{g_i \cdot g_i} \tag{4.4}$$

$$\gamma_i = \frac{(g_{i+1} - g_i) \cdot g_{i+1}}{g_i \cdot g_i} \tag{4.5}$$

If the vicinity of the minimum has the shape of a long, narrow valley, the minimum is reached in far fewer steps than would be the case using the *steepest descent* method, which makes use of minus of the local gradient as the search direction.

In this study, the line minimization to find the step size λ that minimizes $f(x + \lambda h)$ at every iteration step is done by using the *golden section search* algorithm. This is an elegant and robust method of locating a minimum of a line function by bracketing it with three points: if we can find three points a, b, and c where $f(a) > f(b) < f(c)$ then there must exist at least one minimum point in the interval (a, c). The points a, b, and c are said to *bracket* the minimum. This algorithm involves evaluating the function at some point x in the larger of the two intervals (a, b) or (b, c). If $f(x) < f(b)$ then x replaces the midpoint b and b becomes an end point. If $f(x) > f(b)$ then b remains the midpoint with x replacing one of the end points. Either way the width of the bracketing interval will reduce and the position of the minimum will be better defined. The procedure is then repeated until the width achieves a desired tolerance. It can be shown that if the new test point, x, is chosen to be a proportion $(3-\sqrt{5})/2$ (hence Golden Section) along the larger sub-interval, measured from the mid-point b, then the width of the full interval (a, c) will reduce at an optimal rate.

In the absence of an adjoint formalism in COSTA a numerical approximation was used to compute the gradient of the cost function.

4.2.1 Using the Conjugate gradient method

In this COSTA-method both the Fletcher-Reeves and Polak-Ribiere methods are implemented. For comparison purpose, we also implement a steepest-descent method. The user can choose the method to use by specifying the field method *id* in the input file, where 1 refers to Fletcher-Reeves, 2 to Polak-Ribiere, and 3 to steepest-descent.

The stopping criteria are the maximum number of iterations, the tolerance for the step size λ as described in subsection 4.2 and the tolerance for the local gradient. When the line-minimization does not move the parameters significantly to the new ones, the cost function may have reached its (local) minimum. On the other hand, the gradient around the (local) minimum is also close to zero. Hence it is possible to use the norm of the gradient vector as one of the stopping criteria. The same tolerance is also used within the golden section search routine.

Besides specifying the method to use and the stopping criteria parameters, the user also needs to specify the initial guess of the parameters to calibrate as well as the initial two points AX and BX for the golden section search minimization.

Since the method requires the computation of the gradient, while there are no adjoint models available, the gradient of the cost function is computed with finite difference. This is fine for small number of parameters like the ones used in this study. However it will become computationally expensive for many parameters. The size of the perturbation, *delta*, must also be specified in the input file.

In our implementation, the gradient is also stored as COSTA state to make it easier to assign values between the variables representing model parameters to calibrate. The norm of the gradient is computed using the function cta_state_nrm2.

Besides the state vector operations, like scaling and adding, in this method we also need to compute the dot product between two state vectors. This can easily be done by using the function cta_state_dot.

4.2.2 The configuration of the CG method

- `simulation_span`: the overall timespan to run the model
- `maxit`: the maximum number of iterations
- `tol_step, tol_grad`: Stopping criterium according to the step size or the gradient norm
- `AX, BX`: Starting point for golden section search
- `delta`: Perturbation size
- `iteration`: Number of iterations and convergence tolerance
- `model`: Model configuration for the initial model

4.2.3 XML-example

```
<parameter_calibration>
  <CTA_TIME id="simulation_span" start="0" stop="50.0" step=".1"/>
  <iteration maxit="160" tol_step="0.0001" tol_grad="0.0001">
  </iteration>
  <method id="1" AX="0.0" BX="0.02" delta="1E-8">
  <!-- id=1: Fletcher-Reeves, id=2: Polak-Ribiere, id=3: Steepest-descent-->
  </method>
  <model>
     <modelbuild_sp>
     <functions>
        <create>oscillparam_create</create>
        <covariance>oscillparam_covar</covariance>
        <getobsvals>oscillparam_obs</getobsvals>
        <compute>oscillparam_compute</compute>
     </functions>
     <model>
        <parameters avg_t_damp="8.5" avg_omega="12.9">
        </parameters>
     </model>
     </modelbuild_sp>
  </model>
  <paramstd param1="3" param2="3">
  </paramstd>
</parameter_calibration>
```

4.3 Parameter estimation with the LBFGS method

For the problem of minimizing a multivariable function quasi-Newton methods are widely employed. These methods involve the approximation of the Hessian (or its inverse) matrix of the function. The LBFGS (Limited memory-Broyden-Fletcher-Goldfarb-Shanno) method is basically a method to approximate the Hessian matrix in the quasi-Newton method of optimization. It is a variation of the standard BFGS method, which is given by (Nocedal (1980); Byrd et al. (1994))

$$x_{i+1} = x_i - \lambda_i H_i g_i, i = 0, 1, 2, \cdots \tag{4.6}$$

where λ_i is a steplength, g_i is the local gradient of the cost function, and H_i is the approximate inverse Hessian matrix which is updated at every iteration by means of the formula

$$H_{i+1} = V_i' H_i V_i + \rho_i s_i s_i' \tag{4.7}$$

where

$$\rho_i = \frac{1}{y_i' s_i} \tag{4.8}$$

$$V_i = I - \rho_i y_i s_i' \tag{4.9}$$

and

$$s_i = x_{i+1} - x_i \tag{4.10}$$

$$y_i = g_{i+1} - g_i \tag{4.11}$$

Using this method, instead of storing the matrices H_i, one stores a certain number of pairs, say m, of pairs $\{s_i, y_i\}$ that define them implicitly. The product of $H_i g_i$ is obtained by performing a sequence of inner products involving g_i and the m most recent vector pairs $\{s_i, y_i\}$ to define the iteration matrix.

Like in conjugate gradient methods, the line minimization for determining λ in equation (4.6) is implemented by using the Golden Section Search algorithm.

4.3.1 Using the LBFGS method

In the implementation of LBFGS method we use COSTA matrices to store the s_i and y_i vectors in equation (4.10) and (4.11). These vectors are stored as the column of the COSTA matrices. The elimination of the oldest vectors, however, requires the combination of a function for getting a column of a COSTA matrix and another function for setting the column. The latter function is already available in the present COSTA, i.e. cta_matrix_setcol. However, the former one is not yet available. We resorted to worked around this by creating a loop consisting a combination of cta_matrix_getval and cta_vector_setvals for performing this operation.

Vectors s_i and y_i are stored as a COSTA state for the same reason as in the implementation of the conjugate gradient and simplex methods. The operations with a COSTA matrix, however, require the variable to be stored as a COSTA vector. Therefore at some points in the program we were required to use cta_state_getvec and cta_state_setvec to get the values of a COSTA state and assign it to a COSTA vector and vice versa.

4.3.2 The configuration of the LBFGS method

Like in the conjugate gradient method, the implemented LBFGS method requires maximum number of iterations, step-size tolerance, tolerance for gradient, and the perturbation size, *delta*, for computing the gradient. However for using this method the user also needs to specify *nstore*, which is the number of vectors s_i and y_i to store.

- `simulation_span`: the overall timespan to run the model

- `output`: output specification
- `iteration`: Number of iterations and convergence tolerance, `maxln` is the maximum number of iteration steps in the line search part.
- `model`: Model configuration for the initial model
- `method`: Method configuration. `nstore` specifies the storage size, i.e. max number of vector s and y to store. `c1` and `c2` are the parameters used in the 1st and 2nd Wolfe conditions (these are used in the line search part). `delta` specifies the perturbation size in computing the gradient.

4.3.3 XML-example

```
<parameter_calibration>
  <!-- Simulatie timespan en stapgrootte via CTA_Time -->
  <CTA_TIME id="simulation_span" start="0" stop="50.0" step=".1"/>
  <output>
      <filename>results_oscill.m</filename>
     <CTA_TIME id="times_mean" start="0" stop="50.0" step="1"/>
  </output>
  <iteration maxit="10" maxln="20" tol_step="0.0001" tol_grad="0.0002">
  </iteration>
  <method nstore="3" c1="1E-4" c2="0.5" delta="1E-8">

  </method>
     <model>
       <xi:include href="models/lorenz_sp.xml"/>
     </model>
</parameter_calibration>
```

4.4 Parameter estimation with Dud

Dud (Doesn't use derivative) is one of the optimization algorithms, which do not use any derivative of the function being evaluated. It can be seen as a Gauss-Newton method, in the sense that it transforms the nonlinear least square problem into the well-known linear square problem. The difference is that instead of approximating the nonlinear function by its tangent function, the Dud uses an affine function for the linearization. For N calibration parameters, Dud requires $(N + 1)$ set of parameters estimates. The affine function for the linearization is formed through all these $(N + 1)$ guesses. Note that the affine function gives exact value at each of the $(N + 1)$ points. The resulting least square problem is then solved along the affine function to get a new estimate, whose cost is smaller than those of all other previous estimates. If it does not produce a better estimate, the Dud will perform different steps, like searching in opposite direction and/or decreasing searching-step, until a better estimate is found. Afterwards, the estimate with the largest cost is replaced with the new one and the procedure is repeated for the new set of $(N+1)$ estimates. The procedure is stopped when one of the stopping criteria is fulfilled.

4.4.1 Dud Algorithm

Suppose we have a numerical model with a vector of p uncertain parameters, say $\mathbf{x} \in \mathbb{R}^p$. Let $\mathbf{y} \in \mathbb{R}^n$ be a set of n data points and let $f_i : \mathbb{R}^p \to \mathbb{R}$ be the model prediction corresponding to the i^{th} data point. The goal is to find a vector of parameters $\mathbf{x}^{opt} \in \mathbb{R}^p$ that minimizes the following costfunction:

$$(\mathbf{x}) = [\mathbf{y} - \mathbf{f}(\mathbf{x})]^T [\mathbf{y} - \mathbf{f}(\mathbf{x})]. \tag{4.12}$$

The idea of the Dud algorithm is similar to that of the Gauss-Newton algorithm. The key difference is that the Dud algorithm uses an approximation of gradients instead of the exact gradients. In short the Dud algorithm works as follows.

The memory of the algorithm always contains $p+1$ estimations of the optimal parameter set and their corresponding model predictions. First the model functions $f_i(\mathbf{x})$ are linearized based on interpolation between the model predictions that are stored in memory. Then the linearized version of equation 4.12 is solved, yielding a new estimation of the optimal parameter set. Assuming the new estimation is better than all previous ones, it replaces the worst estimation in the memory of the algorithm. This process is repeated until the best estimation is sufficiently close to the optimal parameter set.

Let us now formulate the algorithm in more detail. Let $\mathbf{x}^1, \dots \mathbf{x}^{p+1}$ and $\mathbf{f}(\mathbf{x}^1), \dots, \mathbf{f}(\mathbf{x}^{p+1})$ be the parameter estimations and corresponding model predictions that are stored in memory. The first step is to fill this memory with $p+1$ initial guesses. These guesses are sorted according to the value of their respective costfunctions $S(\mathbf{x}^i)$, meaning $\mathbf{x}^1$ is the parameter set with the highest costfunction and $\mathbf{x}^{p+1}$ the one with the lowest.

Then the model functions $f_i(\mathbf{x})$ are linearized by interpolation between the model predictions that are stored in memory. Let us denote the linearized model functions by $l_i(\mathbf{x})$. Then

$$l_i(\Delta\mathbf{x}) = f_i(\mathbf{x}^{p+1}) + M\Delta\mathbf{x} \tag{4.13}$$

with $\Delta\mathbf{x} = \mathbf{x} - \mathbf{x}^{p+1}$ and M the approximation of the Jacobian matrix by interpolation. Note that $M = FP^1$ where F is the matrix with j^{th} column equal to $\mathbf{f}(\mathbf{x}^j) - \mathbf{f}(\mathbf{x}^{p+1})$ and P the matrix with j^{th} column equal to $\mathbf{x}^i - \mathbf{x}^{p+1}$.

Then $\mathbf{f}(\mathbf{x}) = \mathbf{l}(\mathbf{x})$ is substituted into cost equation 4.12. Let us call the linearized costfunction $Q(\mathbf{x})$. By solving $Q'(\Delta\mathbf{x}) = 0$ we find that for the optimal value of $\Delta\mathbf{x}$ holds

$$M^T M \Delta\mathbf{x} = M^T \left[\mathbf{y} - \mathbf{f}(\mathbf{x}^{p+1})\right] \tag{4.14}$$

yielding a new parameter estimation $\mathbf{x}^* = \mathbf{x}^{p+1} + \Delta\mathbf{x}$. If $\mathbf{x}^*$ has a lower costfunction than $\mathbf{x}^{p+1}$, the worst estimate $\mathbf{x}^1$ is tossed out of the memory, making place for the new estimation. The elements in the memory are again sorted according to their costfunctions, so $\mathbf{x}^{p+1} = \mathbf{x}^*$, $\mathbf{x}^p = \mathbf{x}^{p+1}$ etc.

It may however happen that the new estimation $\mathbf{x}^*$ is not better than one or more of the previous estimations. In this case a line search is done. A better estimation is searched in the direction from $\mathbf{x}^{p+1}$ to $\mathbf{x}^*$, so on the line

$$\mathbf{r}(\epsilon) = \mathbf{x}^{p+1} + \epsilon(\mathbf{x}^* - \mathbf{x}^{p+1}) \tag{4.15}$$

The step size $\epsilon \in \mathbb{R}$ is iteratively being reduced until a step size ϵ^* is found for which $\mathbf{r}(\epsilon^*)$ has a lower costfunction than $\mathbf{x}^{p+1}$. Note that ϵ^* may very well be negative. The new estimation $\mathbf{r}(\epsilon^*)$ is then stored into memory as before.

4.5 Dud with constraints

The parameters $x_i, i \in 1, \ldots, p$ in the calibration process sometimes represent phyisical parameters that should for instance be positive or, based on measurements, need to lie in a specific interval. Due to the linearization process in the Dud algorithm it may very well happen that at some point the Dud algorithm proposes a parameter x_j which does not belong to an expected interval $[a_j, b_j]$. This could blow up the model computation, but it may also be the case that the model automatically corrects such an unexpected value to a value within the expected interval. In the latter case, the Dud algorithm may be ruined. To prevent such unpredicted behaviour, the possibilty to add a set of constraints to the minimization problem in the Dud algorithm was created.

The method of quadratic programming was used for this. First note that minimizing $Q(\Delta\mathbf{x})$ is equivalent to minimizing

$$\begin{aligned} q(\Delta\mathbf{x}) &= \Delta\mathbf{x}^T M^T M \Delta\mathbf{x} 2 \Delta\mathbf{x}^T M^T \left[\mathbf{y} - \mathbf{f}(\mathbf{x}^{p+1})\right] \\ &= \Delta\mathbf{x}^T G \Delta\mathbf{x} - 2\Delta\mathbf{x}^T \mathbf{c} \end{aligned} \tag{4.16}$$

where $G = M^T M$ and $\mathbf{c} = M^T \left[\mathbf{y} - \mathbf{f}(\mathbf{x}^{p+1})\right]$.

To solve this minimization problem with inequality constraints, we need to know how to solve it with equality constraints first. Suppose for now that we have a set of m equality constraints such that $D\mathbf{x} = d$, with $D \in \mathbb{R}^{m\times p}$ and d$\in \mathbb{R}^m$. We want to solve the following minimization problem

$$\begin{aligned} \min_x \quad & q(\mathbf{x}) \\ \text{subject to} \quad & D\mathbf{x} = d \end{aligned} \tag{4.17}$$

A vector $\lambda^* \in \mathbb{R}^m$ of Lagrange multipliers exists, such that for the solution $\mathbf{x}^*$ holds

$$\begin{bmatrix} G & -D^T \\ D & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x}^* \\ \lambda^* \end{bmatrix} = \begin{bmatrix} -\mathbf{c} \\ \mathbf{d} \end{bmatrix} \tag{4.18}$$

This system can be rewritten in a form that is useful for computations by expressing $\mathbf{x}^*$ as $\mathbf{x}^* = \mathbf{x} + \mathbf{s}$, where $\mathbf{x}$ is some estimate of the solution and $\mathbf{s}$ is the desired step. Rearranging equation 4.18 yields

$$\begin{bmatrix} G & D^T \\ D & 0 \end{bmatrix} \begin{bmatrix} -\mathbf{s} \\ \lambda^* \end{bmatrix} = \begin{bmatrix} \mathbf{g} \\ 0 \end{bmatrix} \tag{4.19}$$

where $\mathbf{g} = \mathbf{c} + G\mathbf{x}$ and $\mathbf{s} = \mathbf{x}^* - \mathbf{x}$.

Now let us look at how to handle inequality constraints. We want to solve the following problem

$$\begin{aligned} \min_{\Delta x} \quad & q(\Delta\mathbf{x}) \\ \text{subject to} \quad & A\Delta\mathbf{x} \geq \Delta\mathbf{b} \end{aligned} \tag{4.20}$$

where $\Delta\mathbf{b} = \begin{bmatrix} \mathbf{a} - \mathbf{x}^{p+1} \\ \mathbf{x}^{p+1} - \mathbf{b} \end{bmatrix}$ and $A = \begin{bmatrix} I \\ -I \end{bmatrix}$ with $I \in \mathbb{R}^{p\times p}$ the identity matrix.

We find the solution $\Delta\mathbf{x}*$ iteratively with initial guess $\Delta\mathbf{x} = 0$.

First Iteration
First we ignore all constraints given by equation 4.20, yielding an optimal step $\mathbf{s}$ which satisfies $G\mathbf{s} = \mathbf{g}^0$ where $\mathbf{g}^0 = \mathbf{c} + G\Delta\mathbf{x}^0$.

If $A(\mathbf{s} + \Delta\mathbf{x}^0) \geq \Delta\mathbf{b}$, the constraints are satisfied and we are done with $\Delta\mathbf{x}^* = \mathbf{s} + \Delta\mathbf{x}^0$. If not, we find the constraint i for which $A_i(\mathbf{s} + \Delta\mathbf{x}^0) - \Delta b_i$ is smallest. Here A_i denotes the i_{th} row of the matrix A. By choosing an $\alpha^0 \in [0, 1]$ such that $A_i(\alpha^0\mathbf{s} + \Delta\mathbf{x}^0) = \Delta b_i$ and setting $\Delta\mathbf{x}^1 = \alpha^0\mathbf{s} + \Delta\mathbf{x}^0$ we ensure that $\Delta\mathbf{x}^1$ satisfies all constraints. The i^{th} constraint is then added to the so-called *working set* $\mathbf{W}^1$. The working set $\mathbf{W}^k$ contains all constraints which are active (i.e. act as equality constraints) at the k^{th} iteration.

k^{th} Iteration
We solve the following problem

$$\begin{aligned} \min_{\mathbf{s}} \quad & \mathbf{s}^T G\mathbf{s} + \mathbf{s}^T\mathbf{g}^k \\ \text{subject to} \quad & A_i\mathbf{s} = 0, i \in \mathbf{W}_k \end{aligned} \tag{4.21}$$

If $\mathbf{s} = 0$, we check if all Lagrange multipliers λ_i are positive. If so, we are done with $\Delta\mathbf{x}^* = \Delta\mathbf{x}^{k-1}$. If not, we remove the constraint corresponding to the smallest Lagrange multiplier from working set $\mathbf{W}_{k-1}$, yielding a new working set W_k. Setting $\Delta\mathbf{x}^k = \Delta\mathbf{x}^{k-1}$ we start a new iteration.

If $\mathbf{s} \neq 0$, we check wether $\mathbf{s} + \Delta\mathbf{x}^{k-1}$ satisfies the inequality constraints that are not in working set $\mathbf{W}^{k-1}$. If so, we set $\Delta\mathbf{x}^k = \mathbf{s} + \Delta\mathbf{x}^{k-1}$ and start the next iteration. If not we find the constraint $j \notin \mathbf{W}^{k-1}$ for which $A_j(\mathbf{s} + \Delta\mathbf{x}^{k-1}) - \Delta b_j$ is smallest and choose $\alpha^k \in [0, 1]$ such that $A_j(\alpha^k\mathbf{s} + \Delta\mathbf{x}^{k-1}) = \Delta b_j$. We add this constraint to the working set $\mathbf{W}_{k-1}$, yielding an updated working set $\mathbf{W}_k$. We set $\Delta\mathbf{x}^k = \alpha^k\mathbf{s} + \Delta\mathbf{x}^{k-1}$ and start the next iteration.

Chapter 5

Data assimilation methods available in OpenDA

Origin:	CTA memo200802
Last update:	00-0000

5.1 Introduction to Data assimilation

The terminology of 'data assimilation' originates from the field of meteorology where in 1950's and 1960's new methods were developed to improve weather forecasts. Due to rapid development of computers in the 1950's the applications of large numerical models to weather forecasting became possible. In this early phase the initial state of the model was estimated directly from the measurement. The numerical model was then used to produce the forecast. It was soon recognized that the forecasts could be improved if the initial state was not only based on the measurements but also on the forecast produced by the previous model run. The first data assimilation method was direct insertion which was based on replacement of a model variable by its measured value. Although it was simple to implement, it suffered from the lack of smoothness in the assimilation results. Much of the research at the time was concentrated on finding a proper way to introduce the measurements without introducing oscillations.

In recent years more and more complex methods are being used. These methods are either based on the minimization of a criterion or on statistical principles. This difference divides the data assimilation methods into two classes: the variational and the sequential methods.

The following data assimilation methods are implemented in OpenDA:

- Reduced-rank square-root filter (RRSQRT)
- Ensemble KF (EnKF)
- Ensemble Square-Root filter (EnSRF)

- Complementary Orthogonal subspace Filter For Efficient Ensembles (COFFEE)
- Steady State KF
- Particle Filter
- 3DVar

5.1.1 Variational methods

Variational methods aim at adjusting a model solution to all the observations available over the assimilation period (Talagrand (1997)). In variational methods, one first defines a scalar function which, for any model solution over the assimilation interval, measures the "distance" or "misfit" between that solution and the available observations. The so-called *cost function* will typically be a sum of squared differences between the observations and the corresponding model values. One will then look for the model solution that minimizes the cost function.

A common use of variational method in meteorology is to obtain an optimal initial condition for a model forecast. A suitable cost function for this is the following:

$$J(x_o) = \sum_{k=1}^{N} (y^o(k) - Hx(k))R^{-1}(y^o(k) - Hx(k))' \tag{5.1}$$

where x_o is the initial value of the variable to be determined, $x(k)$ the variable at time t_k, H the observation operator, $y^o(k)$ the observation at time t_k and R the representation covariance. The optimal initial condition, x_o, is the one that minimizes J.

If a suitable initial state x_o has been obtained, the analysed states are formed with a model forecast:

$$x^a_o = x_o \tag{5.2}$$

$$x^a(k) = M[x^a(k-1)] \tag{5.3}$$

for $k = 1, \cdots, N$. The final analysed state x^a_N is optimised given data from spatial different locations and from different times in the interval (t_o, t_N). This approach is then referred to as *4D-VAR*.

The minimization of the cost function is often based on quasi-Newton methods. These methods require computation of the gradient of the cost function. In most situations, it is impossible to establish explicit analytical expressions for the gradient. It is possible to numerically and approximately determine the gradient through explicit finite perturbations of the initial state. But this would be much too costly for practical implementation since it requires to compute the cost function, i.e. to effectively integrate the model over the assimilation period, as many times as there are independent components in the initial states. Therefore to compute the gradient efficiently an adjoint model should be used.

5.1.2 Sequential Methods

While variational method is based on minimization of the cost function within a time interval, sequential method assimilates the data each time the observation becomes available. In sequential method the adjusted model solution is expressed as a linear combination of the forecast state and the data elements following the equation:

$$x^a(k) = x^f(k) + K(k)(y^o(k) - Hx^f(k)) \tag{5.4}$$

Here $x^f(k)$ represents the forecast state, while $x^a(k)$ the analysis state, i.e. the adjusted state. The gain matrix $K(k)$ describes how elements of the state should be adjusted given the difference between the measurement and the forecast. Different methods are available to determine K.

One of the popular sequential methods is the *optimal interpolation* (Daley (1991). This method uses a gain matrix based on an empirical covariance function or matrix. The basic assumption is that both the forecast and the observation error are normally distributed with mean zero and known covariances. The idea of optimal interpolation is to set the analysed state to the conditional mean of the true state given the observations, $x^a(k) = E[x^t(k)|y^o(k)]$. Application of Bayes theorem to Gaussian distribution shows that this could be achieved with a linear gain:

$$x^a(k) = x^f(k) + K(k)(y^o(k) - H'x^f(k)) \tag{5.5}$$

$$K(k) = P^f(k)H[H'P^f(k)H + R(k)]^{-1} \tag{5.6}$$

The gain matrix K in equation (5.6) is known as the *conditional mean gain* or the *minimal variance gain*. A problem is how to choose suitable covariance matrices P and R. In this method the user needs to specify the error covariance at each assimilation time.

Another development in this class is the Kalman filtering. The Kalman filter can be seen as an extension of the optimal interpolation scheme, accounting for the evolution of errors from previous times. The target of the Kalman filter is to obtain a distribution for the true state in terms of a mean $\hat{x}$ and covariance P, given the model and the measurements. Like in the optimal interpolation method, the Kalman filter also assumes normally distributed forecast and observation errors. The Kalman filter is originally derived for linear systems, which in state-space form can be written as:

$$x(k+1) = M(k)x(k) + \eta(k) \tag{5.7}$$

$$y(k) = H(k)x(k) + \nu(k) \tag{5.8}$$

where x is the system state, A the linear model operator, $\eta \sim N(0, Q)$ the system noise, y the predicted observation, H the observation operator, and $\nu \sim N(0, R)$ the observation error. The Kalman filter algorithm consists of two steps:

1. Forecast step:

$$x^f(k+1) = M(k)x^a(k) \tag{5.9}$$

$$P^f(k+1) = M(k)P^f(k)M'(k) + Q(k) \tag{5.10}$$

2. Analysis step:

$$x^a(k) = x^f(k) + K(k)(y^o(k) - H(k)x^f(k)) \tag{5.11}$$

$$P^a(k) = (I - K(k)H(k))P^f(k) \tag{5.12}$$

$$K(k) = P^f(k)H(k)(H'(k)P^f(k)H(k) + R(k))^{-1} \tag{5.13}$$

5.2 Data assimilation with the RRSQRT method

The Kalman filter gives optimal estimates of x and P for linear models. The main problem of applying the Kalman filter directly to environmental models is the computation of the covariance matrix P. Since such models usually have a big number of states (e.g. $O(10^4)$) the covariance will also become very big, which causes very expensive computational costs or even the impossibility to compute. Another problem is that the real life model is usually nonlinear. Therefore methods are proposed and developed to modify the Kalman filter to solve these difficulties. Two most popular algorithms are the *reduced-rank square-root* filter (Verlaan and Heemink (1997)) and the *ensemble Kalman filter* (Evensen (1994)).

The reduced-rank square-root (RRSQRT) filter algorithm is based on a factorization of the covariance matrix P of the state estimate according to $P = LL'$, where L is a matrix with the q leading eigenvectors l_i (scaled by the square root of the eigenvalues), $i = 1, ..., q$, of P as columns. The algorithm can be summarized as follows.

1. Initialization

$$[L^a(0)] = [l_1^a(0), \cdots, l_q^a(0)] \tag{5.14}$$

where l_i^a denotes the q leading eigenvectors of the initial covariance matrix P_o.

2. Forecast step

$$x^f(k) = M[x^a(k-1)] \tag{5.15}$$

$$l_i^f(k) = \frac{1}{\epsilon}\{M[x^a(k-1) + \epsilon l_i^a(k-1)] - M[x^a(k-1)]\} \tag{5.16}$$

$$\tilde{L}^f(k) = [l_1^f(0), \cdots, l_q^f(0), Q(k-1)^{1/2}] \tag{5.17}$$

$$L^f(k) = \Pi^f(k)\tilde{L}^f(k) \tag{5.18}$$

where ϵ represents a perturbation, often chosen close to 1, $\Pi^f(k)$ is a projection onto the q leading eigenvectors of the matrix $\tilde{L}^f(k)\tilde{L}^f(k)'$. Note that here M, the model operator, need not be linear.

3. Analysis step

$$P^f(k) = L^f(k)L^f(k)' \tag{5.19}$$

$$K(k) = P^f(k)H(k)'[H(k)P^f(k)H(k)' + R(k)]^{-1} \tag{5.20}$$

$$x_k^a = x^f a_k + K_k[y_k^o - H_k x_k^f] \tag{5.21}$$
$$\tilde{L}^a(k) = \{[I - K_k H_k]L_f, K_k R_k^{1/2}\} \tag{5.22}$$
$$L^a(k) = \Pi^a(k)\tilde{L}^a(k) \tag{5.23}$$

where $\Pi^a(k)$ is a projection onto the q leading eigenvectors of the matrix $\tilde{L}^a(k)\tilde{L}^a(k)'$. This reduction step is again introduced to reduce the number of columns in L_k^a to q in $\tilde{L}^a(k)$.

5.2.1 Using the RRSQRT method

5.2.2 The configuration of the RRSQRT method

- `simulation_span`: the overall timespan to run the model
- `output`: output specification
- `modes`: The number of modes
- `mode`: Model configuration

5.2.3 XML-example

```
<?xml version="1.0" encoding="UTF-8"?>
<costa xmlns:xi="http://www.w3.org/2001/XInclude">
  <!-- Observations used for assimilation -->
  <CTA_SOBS id="obs_assim" database="obs_lorenz.sql"/>

  <!-- Used model class -->
  <CTA_MODELCLASS id="modelclass"  name="CTA_MODBUILD_SP" />

  <!-- Filter configuration -->
  <method name="lbfgs">
  <parameter_calibration>
    <!-- Simulatie timespan en stapgrootte via CTA_Time -->
    <CTA_TIME id="simulation_span" start="0" stop="50.0" step=".1"/>
    <output>
        <filename>results_oscill.m</filename>
       <CTA_TIME id="times_mean" start="0" stop="50.0" step="1"/>
    </output>
    <iteration maxit="10" maxln="20" tol_step="0.0001" tol_grad="0.0002">
    </iteration>
    <method nstore="3" c1="1E-4" c2="0.5" delta="1E-8">
    <!-- nstore specifies the storage size, i.e. max number of vector s and y to store
    <!-- c1 and c2 are the parameters used in the 1st and 2nd Wolfe conditions-->
    <!-- delta specifies the perturbation size in computing gradient-->
    </method>
       <model>
         <xi:include href="models/lorenz_sp.xml"/>
       </model>
  </parameter_calibration>
  </method>
</costa>
```

5.3 Data assimilation with the Ensemble method

While the RRSQRT represents the covariance matrix P based on the first q leading eigenvectors, the ensemble Kalman filter (EnKF) is based on a representation of the probability density of the state estimate by a finite number N of randomly generated system states. The EnKF algorithm can be summarized as follows.

1. Initialization

   An ensemble of N states $\xi_i^a(0)$ are generated to represent the uncertainty in x_o.

2. Forecast step

$$\xi_i^f(k) = M[\xi_i^a(k-1)] + \eta_i(k-1) \tag{5.24}$$

$$x^f(k) = \frac{1}{N}\sum_{i=1}^{N} \xi_i^f(k) \tag{5.25}$$

$$E^f(k) = [\xi_1^f(k) - x^f(k), \cdots, \xi_N^f(k) - x^f(k)] \tag{5.26}$$

3. Analysis step

$$P^f(k) = \frac{1}{N-1} E^f(k)E^f(k)' \tag{5.27}$$

$$K(k) = P^f(k)H(k)'[H(k)P^f(k)H(k)' + R(k)]^{-1} \tag{5.28}$$

$$\xi_i^a(k) = \xi_i^f(k) + K(k)[y^o(k) - H(k)\xi_i^f(k) + \nu_i(k)] \tag{5.29}$$

where the ensemble of state vectors are generated with the realizations $\eta_i(k)$ and $\nu_i(k)$ of the model noise and observation noise processes $\eta(k)$ and $\nu(k)$, respectively.

For most practical problems the forecast equation (5.24) is computationally dominant. As a result the computational effort required for the EnKF is approximately N model simulations. The standard deviation of the errors in the state estimate are of a statistical nature and converge very slowly with the sample size ($\approx N$). Here it should be noted that for many atmospheric data assimilation problems the analysis step is also a very time consuming part of the algorithm.

5.3.1 The configuration of the Ensemble method

- `simulation_span`: the overall timespan to run the model
- `output`: output specification
- `modes`: The number of ensembles
- `model`: The configuration of a model

5.3.2 XML-example

```
<costa xmlns:xi="http://www.w3.org/2001/XInclude">
  <!-- De invoer m.b.t. de observaties -->
  <CTA_SOBS id="obs_assim" database="obs_advec1d.sql"/>

  <!-- Used model class -->
  <xi:include href="models/advec1d_cls.xml"/>

  <!-- De invoer m.b.t. het filter.  -->
  <!-- Dit filter maakt zelf de model-instantiaties aan -->
  <method name="ensemble">
  <ensemble_filter>
    <CTA_TIME id="simulation_span" start="0" stop="5" step="0.0005"/>
    <output>
       <filename>results.m</filename>
       <CTA_TIME id="times_mean" start="0" stop="50" step="0.05"/>
    </output>
    <modes max_id="50">
       <model>
       </model>
    </modes>
  </ensemble_filter>
  </method>
</costa>
```

5.4 Data assimilation with the Ensemble Square-Root method (ENSRF)

There are two fundamental problems associated with the use of EnKF. First is that the ensemble size is limited by the computational cost of applying the forecast model to each ensemble member. The second one is that small ensembles have few degrees of freedom available to represent errors and suffer from sampling errors that will further degrade the forecast error covariance representation. Sampling errors lead to loss of accuracy and underestimation of error covariances. This problem can progressively worsen, resulting in filter divergence.

In ensemble square-root filters (ENSRF), the analysis step is done deterministically without generating any observation noise realization (Tippett et al. (2003); Evensen, 2004). Since no random sample is generated, this extra source of sampling error is eliminated. Therefore, these methods are expected to perform better than the ones with perturbed observations for a certain type of applications.

Dropping the time index k for simplicity, the covariance analysis update of the ENSRF is obtained by rewriting the covariance as

$$P^a = E^a E^{a'} = [I - P^f H'(HP^f H' + R)^{-1} H]P^f \quad (5.30)$$

$$= E^f [I - E^{f'} H'(HE^f E^{f'} H' + R)^{-1} HE^f]E^{f'} \quad (5.31)$$

$$= E^f [I - V D^{-1} V']E^{f'} \quad (5.32)$$

where $V = (HL^f)'$ and $D = V'V + R$. Then from this equation it is clear that the analysis ensemble can be calculated from

$$E^a = E^f X \quad (5.33)$$

where $XX' = (I - VD^{-1}V')$. Therefore one can say that the updated ensemble E^a is a linear combination of the columns of E^f and is obtained by inverting the matrix D and computing a matrix square root X of the matrix $[I - VD^{-1}V']$. Note that X can also be replaced by XU, where U is an arbitrary orthogonal matrix, so that $(XU)(XU)' = XX'$.

As we have seen that in ENSRF the analysis step consists of determining the *transformation matrix*, X. A number of methods are available to compute X. The method implemented in this study is derived as follows:

$$XX' = (I - VD^{-1}V') = (I - V(V'V + R)^{-1}V') \quad (5.34)$$

If we write $R = SS'$ then we can rewrite Equation 5.34 as

$$XX' = \Psi'(\Psi\Psi' + I)^{-1}\Psi) \quad (5.35)$$

where $\Psi = S^{-1}V = S^{-1}HE^f$. When computing the singular value decomposition of Ψ, i.e. $\Psi = \Gamma\Sigma\Lambda'$, equation (5.35) can be written as

$$XX' = (\Gamma\Sigma\Lambda')'((\Gamma\Sigma\Lambda')(\Gamma\Sigma\Lambda')' + I)^{-1}(\Gamma\Sigma\Lambda')) \quad (5.36)$$

$$= \Lambda(I - \Sigma'(\Sigma\Sigma' + I)^{-1}\Sigma)\Lambda' \quad (5.37)$$

$$= (\Lambda\sqrt{(I - \Sigma'(\Sigma\Sigma' + I)^{-1}\Sigma)})(\Lambda\sqrt{(I - \Sigma'(\Sigma\Sigma' + I)^{-1}\Sigma)})' \quad (5.38)$$

Thus, a solution of the transformation matrix X is given by

$$X = \Lambda\sqrt{(I - \Sigma'(\Sigma\Sigma' + I)^{-1}\Sigma)} \tag{5.39}$$

5.4.1 Using the ENSRF method

The user specifies the number of ensembles in the input file. Like in COFFEE filter, the ensembles in ENSRF filter are also stored as instances of COSTA models. An array of COSTA state vectors is also used for representing matrix E in equation (5.33). There is always a decision to make whether to store E as a COSTA matrix or as an array of COSTA state vectors. We inclined sometimes to store it as a COSTA matrix, since the ENSRF algorithm consists of mostly matrix operations. However, since the model state is stored as COSTA state vector it is easier to store E as an array of COSTA state vector. Moreover, there is also a function available in COSTA for performing linear algebra operation between COSTA matrix and COSTA state, i.e. cta_state_gemm. This function is used for example in computing the correction for all ensemble members, which is done by multiplying the transformation matrix and the forecast ensembles.

The implementation of ENSRF filter requires a singular value decomposition (SVD). At the moment there is no COSTA function, which can perform SVD directly to a COSTA matrix nor to an array of COSTA state vectors. Therefore in this study this is done by assigning the values of the COSTA variables to a Fortran array and use the LAPACK function DGESVD to perform the SVD.

This method also requires the square root of observation error covariance $R^{1/2}$. However, as also mentioned in the previous subsection, the function cta_sobs_getcovmat gives the covariance matrix R. Moreover, there is no COSTA function yet available for computing the square root of a square matrix like for example Cholesky decomposition. However since in the example models the observation noise is always independent, we can easily compute $R^{1/2}$ by taking square root of its diagonal elements.

5.4.2 The configuration of the ENSRF method

- `simulation_span`: the overall timespan to run the model
- `output`: output specification
- `modes`: The number of ensembles
- `model`: Model configuration

5.4.3 XML-example

```
<costa xmlns:xi="http://www.w3.org/2001/XInclude">
```

```
  <CTA_SOBS id="obs_assim" database="obs_oscill.sql"/>
  <!-- Used model class -->
  <CTA_MODELCLASS id="modelclass"  name="CTA_MODBUILD_SP" />
  <method name="ensrf">
<ensrf_filter>
  <!-- Simulatie timespan en stapgrootte via CTA_Time -->
  <CTA_TIME id="simulation_span" start="0" stop="5" step="0.0005"/>
    <output>
       <filename>results.m</filename>
       <CTA_TIME id="times_mean" start="0" stop="50" step="0.05"/>
    </output>

     <modes max_id="50">
        <model>
          <xi:include href="models/oscill_sp.xml"/>
        </model>
     </modes>
</ensrf_filter>

  </method>
</costa>
```

5.5 Data assimilation with the COFFEE method

COFFEE (Complementary Orthogonal subspace Filter For Efficient Ensembles) is a hybrid filter, which combines the RRSQRT filter and the EnKF (Heemink et al. (2001)). One problem of the RRSQRT algorithm is that repeated projection on the leading eigenvalues leads to a systematic bias in forecast errors. Because of the truncation, the covariance matrix is always underestimated, which may result in a filter divergence problem. The truncated part of the covariance matrix does not contribute to the improvements of the state estimate. The COFFEE filter attempts to solve this problem by representing the truncated part of the covariance matrix as random ensembles and to add them to the EnKF part. The RRSQRT part acts as a variance reductor for the ensemble filter, thus reducing the statistical errors of the Monte Carlo approach. Moreover, by embedding the reduced-rank filter in an EnKF the covariance is not underestimated, eliminating the filter divergence problems of the reduced-rank approach (also for very small numbers of q).

The COFFEE algorithm can be summarized as follows:

1. Initialization

$$[L^a(0)E^a(0)] = [l_1^a(0), \cdots, l_q^a(0), \xi_1^a(0), \cdots, \xi_N^a(0)] \tag{5.40}$$

where l_i^a denotes the q leading eigenvectors of the initial covariance matrix P_o. The random ensemble members are generated only to represent the truncated part of the covariance matrix $P_o - L^a(0)L^a(0)'$.

2. Forecast step

   The L^a is updated using the RRSQRT update as follows

$$x^f(k) = M[x^a(k-1)] \tag{5.41}$$

$$l_i^f(k) = \frac{1}{\epsilon}\{M[x^a(k-1) + \epsilon l_i^a(k-1)] - M[x^a(k-1)]\} \tag{5.42}$$

$$\tilde{L}^f(k) = [l_1^f(0), \cdots, l_q^f(0), Q(k-1)^{1/2}] \tag{5.43}$$

$$L^f(k) = \Pi^f(k)\tilde{L}^f(k) \tag{5.44}$$

   where ϵ represents a perturbation, often chosen close to 1, $\Pi^f(k)$ is a projection onto the q leading eigenvectors of the matrix $\tilde{L}^f(k)\tilde{L}^f(k)'$.

   The ensemble ξ_i^a is updated using similar equations with the EnKF update

$$\xi_i^f(k) = M[\xi_i^a(k-1)] \tag{5.45}$$

$$x^f(k) = \frac{1}{N}\sum_{i=1}^{N}\xi_i^f(k) \tag{5.46}$$

$$E^f(k) = [\xi_1^f(k) - x^f(k) + \eta_1(k), \cdots, \xi_N^f(k) - x^f(k) + \eta_N(k)] \tag{5.47}$$

   where η_i are the ensembles representing the truncated part of the covariance matrix with $E[\eta_i(k)\eta_i(k)'] = [I - \Pi^f(k)]\tilde{L}^f(k)\tilde{L}^f(k)'[I - \Pi^f(k)]'$.

3. Analysis step

   In the analysis step the gain matrix K is computed using

$$P^f(k) = L^f(k)L^f(k)' + \frac{1}{N-1}E(k)E(k)' \tag{5.48}$$

$$K(k) = P^f(k)H(k)'[H(k)P^f(k)H(k)' + R(k)]^{-1} \tag{5.49}$$

$$x^a(k) = x^f(k) + K(k)[y^o(k) - H(k)x^f(k)] \tag{5.50}$$

$$\tilde{L}^a(k) = \{[I - K(k)H(k)]L_f(k), K(k)R(k)^{1/2}\} \tag{5.51}$$

$$L^a(k) = \Pi^a(k)\tilde{L}^a(k) \tag{5.52}$$

$$\xi_i^a(k) = \xi_i^f(k) + K(k)[y^o(k) - H(k)\xi_i^f(k) + \nu_i(k)] \tag{5.53}$$

5.5.1 Using the COFFEE method

The COFFEE filter requires the user to specify in the input file the number of modes for the RRSQRT part and the number of ensembles for the EnKF part. Moreover, the perturbation δ (i.e. ϵ in equation (5.42)) needs also to be specified.

In this implementation, the modes and ensembles are represented as COSTA model instances. Here each model instance represent one mode or ensemble. This is chosen for supporting parallel computation implementation in the future.

In the implementation of COFFEE filter, the initial matrix $[L^a(0)E^a(0)]$ in equation 5.40 is always assumed to be zero. This however should be extended to accommodate the option where the user can specify this from the input file. Moreover, the size of L is limited to a maximum of 200. This is due to the fact that its dimension varies according to the number of noise parameters in the model as well as to the number of observations. Since we know the number of observations only at the analysis-step and that the number of observations may vary, it is not possible to determine the dimension of L a priori. Furthermore, in the present implementation the computation cost increases with number of observations. More study is required to develop the implementation for solving this problem. It should be noted also that the present implementation does not support covariance localization using Schur product either.

The COFFEE filter also requires the square root of the model error covariance matrix $Q^{1/2}$ in equation (5.43) as well as the observation error covariance matrix $R^{1/2}$ in equation (5.51). The covariance matrices Q and R can be obtained by using the functions cta_model_getnoisecovar and cta_sobs_getcovmat respectively. However since we mostly work with the square root of the covariance matrices, it is better if these functions are modified to give the square root matrices. Note that cta_model_getnoisecovar gives the noise covariance matrix in terms of an array of COSTA state vectors, while cta_sobs_getcovmat gives the observation noise covariance matrix in term of COSTA matrix. This may cause confusion for new programmers working with COSTA. Moreover, it may be better to use a similar name for the two functions. This makes the functions easier to remember.

Note that for general nonlinear models, the additional columns of matrix L^f with matrix $Q^{1/2}$ in equation (5.43) can be computed using finite difference as follows

$$\hat{x}(k) = M[x(k-1), u(k-1), 0] \tag{5.54}$$

$$x_i^n(k) = M[x(k-1), u(k-1), \epsilon Q^{1/2}(:, i)] \tag{5.55}$$

$$l_i^n(k) = \frac{1}{\epsilon}(x_i(k) - \hat{x}(k)) \tag{5.56}$$

$$\tilde{L}^f(k) = [l_1^f(k), \cdots, l_q^f(k), l_1^n(k), ..., l_w^n(k)] \tag{5.57}$$

where M is the general nonlinear model operator, which is a function of the previous time-step state vector x, the input forcing u and the noise η. The state vector $\hat{x}$ is obtained from the deterministic model, i.e. when no noise is present, while x_i^n is from the stochastic model, with the realization $\eta = \epsilon Q^{1/2}(:, i)$, where $Q^{1/2}(:, i)$ refers to the i^{th} column vector of matrix $Q^{1/2}$. The implementation requires a function which can set the model noise realization. However, since there is no such function available yet, it is not possible to implement this in COSTA at the moment.

5.5.2 The configuration of the COFFEE method

- `simulation_span`: the overall timespan to run the model
- `output`: output specification
- `modes`: The number of modes
- `delta`: perturbation, see equation 5.42
- `model`: Model configuration
- `ensembles`: the number of ensembles

5.5.3 XML-example

```
<costa xmlns:xi="http://www.w3.org/2001/XInclude">
  <CTA_SOBS id="obs_assim" database="obs_oscill.sql"/>

  <!-- Used model class -->
  <CTA_MODELCLASS id="modelclass"  name="CTA_MODBUILD_SP" />


  <method name="coffee">
<coffee_filter>
  <CTA_TIME id="simulation_span" start="0" stop=".5" step="0.0005"/>
    <output>
       <filename>results.m</filename>
       <CTA_TIME id="times_mean" start="0" stop="50" step="0.05"/>
    </output>

     <modes max_id="10">
        <delta id="1E-4"> </delta>
        <model>
          <xi:include href="models/oscill_sp.xml"/>
        </model>
     </modes>

     <ensembles max_id="5"> </ensembles>
</coffee_filter>

  </method>
</costa>
```

Chapter 6

OpenDA black box wrapper cookbook

Contributed by: Nils van Velzen
Last update: 00-0000

6.1 What not to expect from this chapter

This chapter is not a thorough reference manual on configuring and using the black box wrapper of OpenDA. The purpose of this chapter is to give you some overview information to get you started making your own black box wrapper for your model. There are many example implementations available and at some point we hope that there will be an user guide available for the black box wrapper explaining all the details on configuration and implementation. This kind of detailed information is not to be found in this chapter. This chapter should be used in combination with the existing examples and the OpenDA course[1] where you already can find a lot of information on the black box wrapper.

6.2 The design of the black box wrapper

Using models in black box form in OpenDA means that data assimilation and model calibration techniques are applied to a model without changing the existing model code. OpenDA and the black box wrapper have no knowledge of the model internals (black box) and only uses the input and output files of the model. The black box implementation of OpenDA can also be used in a slightly different way as well like we do for C# models and EFDC, but this is beyond the scope of this document.

The black box wrapper is an implementation of a stochastic model in OpenDA. The difference with a normal model is that the black box wrapper does not contain any model equations. It is a layer that helps you to link and configure a model to OpenDA using the black box approach. The black box wrapper is illustrated in Figure 6.1, the elements of this figure will

[1]The OpenDA course is available as a separate download from the OpenDA website.

be explained in some more detail in the following sections.

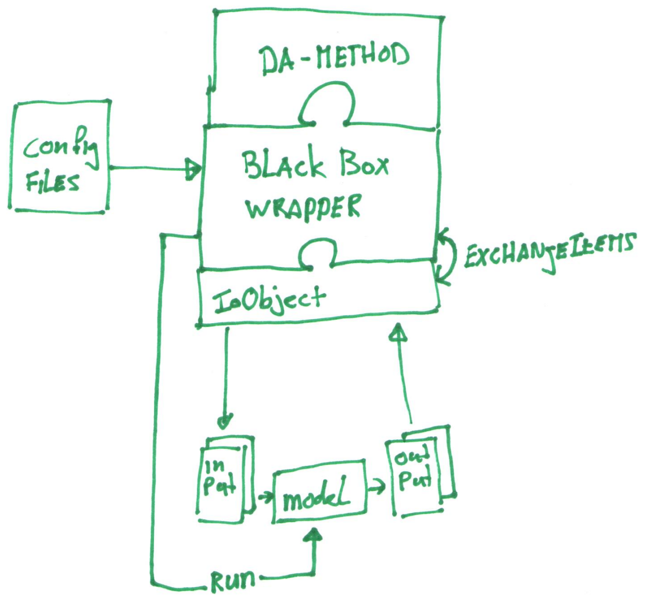

Figure 6.1: Overview of a black box configuration in OpenDA. The model interacts with the generic black box wrapper of OpenDA implementing an OpenDA stochastic model. The black box wrapper builds its state, parameters and predictions using exchangeItems. The exchangeItems contain values that are read from model output files or need to be written to model input files. The creation of the exchangeItems and reading from/writing to files is implemented by one or more ioObjects. The black box wrapper will run the model executable when all input files are prepared.

6.3 ExchangeItems

Values from the input and output files are represented in the OpenDA black box wrapper in the form of exchangeItems. An exchangeItem is a container of values with optionally some meta information. In the most simple form an exchangeItem contains a single value or array and has en ID. ExchangeItems can have three different roles:

- input: the values have been read from one of the output files of the model. These values are used by the algorithm but (updated) values will not be written back to the file.
- output: the values will be computed by the algorithm and will be written to the input file of the model.

- inout: the values are both read and written to a file of the model that is both input as output (typically restart files).

There is one special kind of exchangeItem that is now worth mentioning and that is the timeSeriesExchangeItem. This kind of exchangeItem contains a sequence of values that are all associated to a timestamp (meta information). This kind of exchangeItem is typically used for storing model results that need to be matched to observations.

6.4 Model wrappers

The model uses files for input and output and the black box wrapper uses exchangeItems. Therefore we need to write some code that creates all the exchangeItems and handles the reading and writing of the values in the exchangeItems from and to the model input and output files.

For each file format we need to handle we have to implement a java class that implements the IoObjectInterface interface. This interface only contains three public methods:

- initialize: creates all exchangeItems. The values of the input and inout exchangeItems are read from file.
- getExchangeItems: Returns an array containing all the exchangeItems of this file.
- finish: Write the values of output and inout exchangeItems to file.

OpenDA already contains quite a number of implementations of exchangeItems. In almost all cases you can use one of the existing implementations as container of the values for your model. Useful existing implementations include:

- org.openda.exchange.DoubleExchangeItem: ExchangeItem for a single double
- org.openda.exchange.DoublesExchangeItem: ExchangeItem for an array of doubles
- org.openda.exchange.timeseries.TimeSeries: Timeseries of double values each having a time associated.

Your implementation only needs to create those exchangeItems that you think you will need. In an first implementation you do not need to bother about all possible data in the files. Additional exchangeItems can always be added later when needed.

6.5 Recipe

- Identify the input and output files that contain values you need and determine how these values are stored
- Write an IoObjectInterface class for each file format you need such that you have exchangeItems for all the data you are going to use in OpenDA. Try to use existing implementations of exchangeItems when possible. For inspiration look at available example implementations. Note: Use a "simple" wrapper like reactive_advection_pollution_wrapper from the OpenDA course or model_nemo as inspiration and not one of the very complex wrappers with a lot of support and functionality.
- set up a black box configuration environment (model input and configuration files). Best way is to copy an existing example and replace model input files and change the black box configuration files. The basic ideas of the black box configuration are explained in the OpenDA course.
- Done

Chapter 7

Generation of output with OpenDA

Last update: 03-2014

7.1 Concepts

OpenDA has an advanced output processing system. The main concept can be understood as a publisher-subscriber mechanism. In many places in the code of OpenDA developers have identified pieces of information, that may be of interest to users. Each piece of information gets an 'id', an 'outpuLevel' and 'context'. Separate from offering potentially useful items anywhere in the code, you can configure one or more output-writers and filter the output that you want to collect. The type of ResultWriter selected determines the output format. Different writers, or multiple copies can be selected in one run, eg to split your results in separate files that are easier to analyse.

The current implementation replaces an older one, that is still working for compatibility of older configurations. You may encounter input files with the old syntax. We suggest that you replace them with the new syntax described here. If you really need to use the old format, please consult the xsd-file or xsd-documentation on our website.

7.2 Getting started

We recommend that you start with following defaults at the end of your main oda-file:

```
<resultWriters>
  <resultWriter className="org.openda.resultwriters.MatlabResultWriter">
    <workingDirectory>.</workingDirectory>
    <configFile>results_yourlabel.m</configFile>
    <selection>
      <resultItem outputLevel="Normal" maxSize="1000" />
    </selection>
  </resultWriter>

  <resultWriter className="org.openda.resultwriters.NetcdfResultWriter">
    <workingDirectory>.</workingDirectory>
    <configFile>results_yourlabel_.nc</configFile>
    <selection>
      <resultItem outputLevel="Normal" minSize="1000" />
    </selection>
  </resultWriter>
</resultWriters>
```

This configures two resultwriters: all small items are written to matlab-ascii and the larger items to netcdf. The default outputLevel should be good for a start, but can easily be adapted if needed. Valid outputLevels are: None, Essential, Normal, Verbose and All.

7.3 Advanced filtering

In some cases the simple approach above is not sufficient and more advanced filtering is needed. You should keep in mind that multiple criteria for selection will only select those items that match all criteria. For example:

```
<selection>
  <resultItem outputLevel="Normal" maxSize="1000" />
</selection>
```

selects items that have outputLevel higher or equal to Normal and have a size that is at most 1000. On the other hand selection in separate items are matched if one applies. For example:

```
<selection>
  <resultItem outputLevel="Normal" maxSize="1000" />
  <resultItem outputLevel="Essential" minSize="1001" />
</selection>
```

will add the most important large items compared to the previous example.

The most important section criteria are:

- outputLevel : Hint by the developer about the importance of the output. Valid values are None, Essential, Normal, Verbose and All
- minSize : only items of at least this size are selected
- maxSize : only items of at most this size are selected

- id : only id's matching this selection are written, eg "pred_f". At the moment you can only find out about the id's with a small trial run, where you write all items, or by examining the code.
- context : The items are divided in groups that are intended to help you suppress output or generate more output for parts of the algorithm. For example the analysis-step of the EnKf used context="analysis step". One can also use regular expressions here. For example to select inner-iteration 10 one uses context="inner iteration 10", but to select all inner-iterations one can use context="inner iteration (.*)".

7.4 ResultWriters

In addition to the two most common result-writers, several others exist for more specific purposes. Moreover, you can implement your own result-writer to format the output exactly to your needs. Some existing writers are:

- `org.openda.resultwriters.MatlabResultWriter`: Most common output to matlab ascii format (.m file). These files can be loaded into Matlab or Octave for further analysis or plotting. Since the ascii format is very slow for large data items, the default maximimum size is maxSize=1000.
- `org.openda.resultwriters.NetcdfResultWriter`: Writes to netcdf. Very useful for larger data items, that do not fit well in the Matlab ascii files.
- `org.openda.resultwriters.TextTableWriter`: Simple comma-separated-value table with main results of a calibration run. Can eg be used to load the main results into a spreadsheet program.
- `org.openda.resultwriters.CsvResultWriter`: TODO
- `org.openda.resultwriters.GlueCsvResultWriter`: TODO
- `org.openda.resultwriters.McCsvResultWriter`: TODO

7.5 Finally

Several nice examples can be found in the examples-directory
`public/examples/core/simple_resultwriters`.

Chapter 8

Parallel computing in OpenDA using Threading and Java RMI

Contributed by:	Nils van Velzen
Last update:	03-2014

8.1 Front-end models in OpenDA

Front-end models are a special kind of models in OpenDA. A front-end model implements the OpenDA Stochastic model interface but it does not implement any model equations. A front-end model adds additional functionality on top of an arbitrary OpenDA Stochastic model. This model is called the back end model. In general, all methods of the front-end model are implemented by use of methods of the back-end model.

The front-end models implement generic extension to OpenDA Stochastic models.

In this chapter we will give a description of two of these kind of front-end models. The Thread Stochastic Model adds some parallelism to arbitrary OpenDA stochastic models using threading. The RMI Stochastic Model allows models to be run in a different process and optionally remotely on a remote computer.

8.2 Parallelism using Threading

8.2.1 introduction

Java has a good support for threading. Using threads, a program can make use of all the available computational cores in a computer. Threading is therefore in Java a simple and natural way to implement shared-memory parallelism.

Many data assimilation and calibration algorithms perform a (large) number of model simulation steps

$$x(t+\Delta t) = M(x(t), u(t), p) \tag{8.1}$$

for various state vectors $x(t)$ or parameters p. When these model simulations are independent, we can compute them in parallel.

8.2.2 Thread Stochastic Model

Basic usage and configuration

OpenDA provides a front-end model that parallelizes the compute method of a stochastic model using threading. This model can be used as a front-end to any OpenDA model that implements the compute method in a thread safe way. The maximal number of threads can be specified in order to limit the number of simultaneous compute invocations.

The model factory of the threaded front-end model is

```
org.openda.models.threadModel.ThreadStochModelFactory
```

The configuration file of the `ThreadStochModelFactory` specifies

- maxTheads; the max number of simultaneous compute threads. This is typically set to the max number of cores that are available to the user.
- stochModelFactory; the configuration of the the back-end stochastic model factory.

A typical example of the `ThreadStochModelFactory` configuration looks like

```
<threadConfigstoch>
   <maxThreads>8</maxThreads>
   <stochModelFactory
      className="org.openda.models.lorenz.LorenzStochModelFactory">
      <workingDirectory>model</workingDirectory>
      <configFile>LorenzStochModel.xml</configFile>
   </stochModelFactory>
</threadConfigstoch>
```

NOTE1: The compute method of your model MUST be thread safe. Java models will often be thread safe. Native models are in general not thread safe.

NOTE2: The working directory for the back-end model is NOT relative to the location of the `ThreadStochModelFactory` configuration file but to the ”main” OpenDA configuration file.

advanced features

The basic usage of thread stochastic model only includes specifying the max number of threads and the back-end model factory. There are some additional options that can improve the performance of your parallel application. These advanced options are described in this section. The impact of these options depend heavily on the used model and architecture.

- `sleepTime` (integer) the number of miliseconds sleep between polling whether a thread running the model has finished. When set too small, the polling process can take a significant amount of CPU time slowing down the model runs. The default value of 10 (ms)will work ok for most models.
- `cashState` (boolean) if set "true", a parallel nonblocking getState method is invoked after a parallel compute is completed. This parallel getState method is invoked on a new thread and not limited by the maxThread option. In this way, the data assimilation algorithm already receives a copy of the state while other model instances are still computing. This can be useful when the time of the getState method is dominated by IO (file or network). The state is stored internally until the getState method is invoked.
- `nonBlockingAxpy` (boolean) if set "true", a invocation of the axpyState method will be non-blocking. This option might improve performance when the time of the axpyState method is dominated by IO (file or network).

The XML configuration using these three options looks like:

```
<threadConfig>
   <maxThreads>2</maxThreads>
   <sleepTime>50</sleepTime>
   <cashState>true</cashState>
   <nonBlockingAxpy>true</nonBlockingAxpy>
   <stochModelFactory
    className="org.openda.models.rmiModel.RmiClientStochModelFactory">
      <workingDirectory>./stochModel</workingDirectory>
      <configFile>RmiStochModel2.xml</configFile>
      </stochModelFactory>
</threadConfig>
```

8.3 Remote Method Invocation

8.3.1 Introduction

RMI (Remote Method Invocation) is a way in java to make a connection between objects in various virtual machines (executables). In OpenDA we provide a front-end model that enables us to create and use stochastic models on various executables and computers. Note that RMI allows us to use multiple processes in our application but that computations are not automatically in parallel.

8.3.2 RMI Stochastic Model

The RMI provides the (parallel) computation of model simulation steps in a different virtual machine as the data assimilation method or the model calibration method. This serves two goals:

- multiple model time steps can be computed in parallel (in combination with the Thread model front-end)
- the model can runs on a dedicated machine (server) where the data assimilation method runs locally.

The OpenDA application running the data assimilation algorithm or model calibration algorithm is called the client. This executable will make use of one or more servers, possibly running on remote machines.

The RMI front-end model factory is

`org.openda.models.rmiModel.RmiClientStochModelFactory`

The configuration file of the `RmiClientStochModelFactory` specifies

- serverAddress; The names of the computers running the servers. There are three ways of configuration possible:
  - empty; all servers run on the local host
  - single machine name; all servers run on this machine
  - names of all computers, comma separated. Note that the number of computer names must be the same as the number of specified factoryIDs. The same computer name can be used multiple times when multiple servers run on that computer.
- factoryID; The unique IDs of all the servers, comma separated, implementing the remote models and model factories.
- stochModelFactory; the configuration of the the back-end stochastic model factory. **Important note**: The `workingDirectory` and `configFile` are communicated to the server processes. The use of relative paths is only possible when the servers are started from an appropriate location. Full paths are therefore often a better choice.

A typical example of a `RmiClientStochModelFactory` configuration file is

```
<rmiConfig>
   <serverAddress></serverAddress>
   <factoryID>IRmiIStochModel_1,IRmiIStochModel_2</factoryID>
   <stochModelFactory
      className="org.openda.models.lorenz.LorenzStochModelFactory">
      <workingDirectory>./model</workingDirectory>
      <configFile>LorenzStochModel.xml</configFile>
   </stochModelFactory>
</rmiConfig>
```

8.3.3 RMI Stochastic Model Server

In the previous section we have explained how to use the RMI front-end model from the client side. The other side are the server processes. The java class that implements the server is `org.openda.models.rmiModel.Server`.

Before the servers can be started a special daemon process must be started **`rmiregistry`** on each host computer. This daemon process allows RMI clients and servers to connect. The following example shows a shell script that starts and initialises two servers on local host.

```
#!/bin/sh

# append all jars in opendabindir to java classpath
for file in $OPENDADIR/*.jar ; do
   if [ -f "$file" ] ; then
       export CLASSPATH=$CLASSPATH:$file
   fi
done

rmiregistry &
echo wait 5 seconds for rmiregistry to start and initialize
sleep 5

# start the server with a non-default factory ID
echo starting server
java  -Djava.rmi.server.codebase=file:///$OPENDADIR/ \
      org.openda.models.rmiModel.Server IRmiIStochModel_1 0 2&
java  -Djava.rmi.server.codebase=file:///$OPENDADIR/ \
      org.openda.models.rmiModel.Server IRmiIStochModel_2 1 2&
```

Note the `-Djava.rmi.server.codebase=file:$OPENDADIR` option. This specifies the starting point to the classes that are loaded by the server. The two arguments have the following meaning:

1. (string) ID of this server

2. (int) the sequence number of this server (0,...,number of servers -1)
3. (int) total number of servers

The last two numbers are used to configure the DistributedCounter as will be explained in the following section.

8.3.4 InfiniBand and multiple network cards

Some nodes have multiple network connections. On supercomputers you will typically see that nodes both have an ethernet and an InfiniBand connection. In this case the computer will have two ip addresses and probably two names. The RMI server will use the network connection that corresponds to the $HOSTNAME of the node. This is typically the ethernet connection and not the fast InfiniBand we would like to use. The java option

```
-Djava.rmi.server.hostname="InfiniBandHostname"
```

can be passed to Java when starting the servers. In this way, the servers will use the InfiniBand connection for communication to the algorithm process.

8.3.5 Random numbers and DistributedCounter

When running in parallel you sometimes need to generate ”unique” numbers in the model instances. E.g. when each model instance generates (or reads) its own input and output files and when you want to use some numbering to distinguish them. In order to globally define unique counters we have introduced the DistributedCounter class in OpenDa. When used in a sequential run, a DistributedCounter instance is just a integer counter 0,1,2,3,etc. In parallel runs, the DistributedCounter instance will generate a sequence $i_p + jn_p$ for j=$0, 1, ...$ where i_p denotes the process number and n_p the total number of processes.

A similar issue is the generation of pseudo random numbers. We must be very careful when we generate random numbers in a parallel environment. If the random generators in various threads of processes are all initialised with the same initial seed, we will get wrong results since the same pseudo random numbers will be drawn on the various processes. The DistributedCounter can be used to initialise pseudo random generators. When the user uses one of the noise models from OpenDA this should work correct in parallel since these random generators make use of the DistributedCounter.

8.3.6 Serialization

All arguments (classes) that are passed between the remote model instances need to ”extends Serializable”. If this is not the case Java cannot pack the content and send the object to an other processes. Many of the relevant classes in OpenDA are already Serializable but user provided classes are often not. When a class is not Serializable you will get a run-time error when you try to run in parallel. Fortunately, this is often easy to fix by adding "extends Serializable" to these classes (and subclasses).

8.3.7 Limitations

The RMI front-end model is work in progress. There are therefore some limitations including

- Only java implementations of (Tree-)Vectors can be used because the serialization of native objects is not yet implemented
- We have not done any work/testing on client/server security issues but we did not encounter any problems on the HPC computer systems we have used. There might be some more work to be done for using these tools in more secured environments.

8.4 Parallel computing with multiple executables

The Thread Stochastic model allows parallel computation of model time steps. This parallelism is limited to a single executable and maximized by the amount of cores of a single computer. The RMI Stochastic model allows multiple executables to be used for the model computations. But The RMI model does not by itself support parallelism. However when both front-end models are combined we are able to compute model time steps in parallel using multiple processes and multiple computers. The Thread model is then used as a frond-end to the RMI model which is the front-end of the physical model.

Models that are not thread safe can be parallelized by combining both front end models because at the server side no time steps are computed in parallel.

Chapter 9

Performance monitoring and tuning in OpenDA

Contributed by: Nils van Velzen, VORtech
Last update: 03-2014

9.1 Introduction

Data assimilation and model calibration techniques can be very computationally intensive. A good performance of your OpenDA configuration is therefore often important. OpenDA contains a number of options that can help you to monitor and tune/improve the performance of your configuration. The currently available tools will be presented in this document.

9.2 monitoring

A popular Dutch (scientific) saying is ”meten = weten”, which can be translated into ”to measure = to know”. The idea is simple, if you measure you will know what is going on and that allows you to improve/change a process when needed. For improving and tuning the performance this counts. Before you can start to improve the performance in a sensible way you need to know the performance of the individual parts.

There are many (Java) profiling tools available to measure the performance of your code. Unfortunately these tools will give you information on the time spend in various routines. These timings are useful for programmers with detailed knowledge of the implementation but it is not practical for a general user. If you profile your configuration you want to have information like the time spend in model steps, getting the state vector, interpolating the observations, multiplication with the gain matrix etc. To provide this type of timing information we have introduced the OdaTiming class in OpenDa. Programmers can put these timers in the code to time interesting well defined steps in the algorithm. The timers from OdaTiming have labels and there is a way to define sub timers, all to present the result in a human understandable form.

The internal timing mechanism of OpenDA can be switched on and off in the main configuration file of your OpenDA configuration.

```
<timingSettings doTiming="true"></timingSettings>
```

A timing report file will be created at the end of your run when timing in switched on.

The timers are introduced by the programmers. Therefore not all parts of OpenDA are currently timed. At this moment timers are present in:

- Ensemble Kalman Filter.
- Blackbox model.
- rmiModel and threadModel.

At some points programmers should/can extend the timing for other algorithms and parts of OpenDA as well.

9.3 Precision

The choice of the precision of your computations is an important issue. The most used precisions in scientific computing are double precision (double) and single precision (float). Double precision is sometimes necessary for sufficient accuracy or numerical stability. From the point of performance, single precision is preferable when possible because it halves the storage and communication volume and computations are faster as well.

We have introduced some control over precision in OpenDA. Note the word ”some” since OpenDA cannot force the precision of values stored in the user provided classes since it is an object oriented system. In the central input file you can specify

```
<optimizationSettings productionRun="false" VectorPrecisionIsFloat="true">
</optimizationSettings>
```

The option VectorPrecisionIsFloat=”true” will mark the wish of the user to use single precision. Programmers of OpenDA classes can introduce dual precision support by checking the user selected precision in the global static class OdaGlobSettings.

Currently the standard (java) vector implementation in OpenDA (Vector) is implemented in dual precision. Since many classes like the black box model make use of the standard Vector implementation it already works in many configurations.

Note: you can always use the available timing in OpenDA to spot the impact of the ”VectorPrecisionIsFloat” option. If you do not see any improvement in linear algebra computations it is likely that the class used in your implementation ignores the user precision preference. In that case you can introduce it yourself or try to convince the programmer of that class to do it for you.

9.4 Production runs

Data assimilation methods sometimes produce some diagnostics. The amount of time to compute these diagnostics can be quite large. At this moment an algorithms cannot ask a result writer in advance whether data will be written or not. As a result all diagnostics will always be computed whether the user wants them or not. The user can set the options ”productionRun” in the global class OdaGlobSettings by specifying it in the central input file.

```
<optimizationSettings productionRun="true" VectorPrecisionIsFloat="true">
</optimizationSettings>
```

Programmers of the assimilation methods can use this user provided option to skip some diagnostics part of the computations.

Note: Currently, this option only impacts the behaviour of the Ensemble based methods in OpenDA.

Chapter 10

OpenDA bias aware model

Contributed by: Nils van Velzen, VORtech
Last update: 07-2014

10.1 Bias aware modelling

Data assimilation methods normally assume no bias in the model and the observation errors. In real life this is unfortunately not always the case. There are some mathematical methods available to detect and estimate structural differences between observations and mode (bias). OpenDA contains a wrapper model that enables perform experiments to estimate and the bias between model and observations.

10.2 Algorithm

OpenDA uses state augmentation to estimate the bias. The method is explained in more details in Drécourt et al. (2006). A model is defined in OpenDA according to

$$x^{k+1} = M\left(x^k, u^k, p, w^k\right) \tag{10.1}$$

With the model state x, forcings y, parameters p and noise w. The interpolation operation to compare the model predictions to the observations y^k is defined by

$$Hx^k \tag{10.2}$$

We assume there is a bias b between model prediction Hx and observations y. If we would know this bias we can correct the observations y with b before assimilating.

$$\left[\begin{array}{c} x^{k+1} \\ b^{k+1} \end{array}\right] = \left[\begin{array}{c} M\left(x^k, u^k, p, w^k\right) \\ b^k + n^k \end{array}\right] \tag{10.3}$$

The interpolation operation for this bias correcting model is

$$Hx^k - b^k \tag{10.4}$$

The bias aware model in OpenDA is a generic wrapper model that extents an arbitrary OpenDA model, implementing Equations 10.1 and 10.2 into a model that implements Equations10.3 and 10.4.

10.3 configuration

In order to use the bias aware model, the user specifies the

`org.openda.models.biasAwareObservations.BiasAwareObservationsModelFactory`

in the main OpenDA configuration file. The "real" dynamical model is specified in "child" model in the configuration of the bias aware model.

The model configuration contains of two parts:

1. Definition of the child model (xml-tag `stochModelFactory`)
2. Definition of the bias model (xml-tag `state`

The format of the `stochModelFactory` is exactly the same as in the definition of a stochastic model factory in the main OpenDA configuration file. It contains:

- the attribute `className` (mandatory), specifying the implementation of the model factory
- the tag `workingDirectory` (mandatory), specifying the main directory of the model configuration
- the tag `configFile` (mandatory), file containing the model configuration

The second part with tag `state` defines the augmented state that is used to model the bias. This tag contains:

- the attribute `maxSize` (optional), the size of the augmented state. This attribute can only be left out when all observations are individually specified using the `observation` attributes.
- the attribute `localization` (optional, default="true") If set "true" we assume all elements in the augmented state to be non correlated. If the filter uses localisation, only the single matching observation is used to update each element of the augmented state. If set to "false" no localization is used, all localization weights of the augmented state are set to one.

- the attribute `standard_deviation`. The standard deviation of the random walk noise for a one day period.
- the attribute `checkObservationID` (optional, default="true"). Match elements in the augmented state using the ID of the observations. If set to "false" the algorithm expects that the i-th observation at each assimilation step corresponds to the i-th element of the augmented state.
- the tag `observation` (optional,repetitive).Specification of biases corresponding to individual observations. This attribute has the following attributes:
  - `id` (mandatory) name of the observation device/location
  - `standard_deviation` (mandatory) standard deviation of the random walk noise for a one day period

It is sometimes not known in advance what the id's are of the observations. A trick is to perform a small run, not individually specifying the observations. In the message file you will then find the id's of the observations when they are assigned to an element of the augmented state for the first time.

A typical configuration where all observations have the same bias uncertainty looks like:

```
<BiasAwareModelConfig>
<stochModelFactory
className="org.openda.models.lorenz.LorenzStochModelFactory">
<workingDirectory>.</workingDirectory>
<configFile></configFile>
</stochModelFactory>
<state standard_deviation="3.0"
maxSize="10"  localization="false"  checkObservationID="true">
</state>
</BiasAwareModelConfig>
```

A configuration that individually configures the biases looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<BiasAwareModelConfig>
<stochModelFactory
className="org.openda.models.lorenz.LorenzStochModelFactory">
<workingDirectory>.</workingDirectory>
<configFile></configFile>
</stochModelFactory>
<state localization="false"  checkObservationID="true">
<observation id="point1.waterlevel"
```

```
             standard_deviation="0.01"></observation>
<observation id="point2.waterlevel"
             standard_deviation="0.02"></observation>
<observation id="point3.waterlevel"
             standard_deviation="0.03"></observation>
<observation id="point4.waterlevel"
             standard_deviation="0.04"></observation>
<observation id="point5.waterlevel"
             standard_deviation="0.05"></observation>
</state>
</BiasAwareModelConfig>
```

Chapter 11

Using the modelbuilders in COSTA

Remark: COSTA is incorporated in `OpenDA`, `/public/core/native/src/cta`.

Contributed by:	Nils van Velzen, CTA memo200605
Last update:	03-2014

11.1 Introduction

The COSTA environment makes a number of building blocks available for creating data assimilation and calibration systems. Combining and building new building blocks should be possible with a minimum of effort.

COSTA contains tools for rapidly creating a COSTA model component. These tools are called modelbuilders. The various modelbuilders are described in this document.

11.2 General description of a model

COSTA deals with assimilation methods for simulation models. Simulation models can compute the model state at different time instances.

$$\begin{aligned} \phi(t_0) &= \phi_0, \\ \phi(t_{i+1}) &= A[\phi(t_i), u(t_i), g] \end{aligned} \tag{11.1}$$

with

- ϕ_0 the initial model state,
- $\phi(t)$ the model state at time t,
- A the operator that computes one time-step of the numerical simulation model,
- $u(t)$ the time dependent forcings at time t,

- g the time independent model parameters

The model as stated in Equation 11.1 is a general form. This means that it is not mandatory that all arguments exist in the model. An extreme example is the model, as specified by Equation 11.2 that can be used in a calibration context where an optimal value for g is determined using observed data.

$$\phi = A\left[g\right] \tag{11.2}$$

11.3 SP Model builder

The SP modelbuilder (Single processor) can be used to create sequential (non-parallel) model components. The SP modelbuilder handles the storage and administration of the model-instance specific data. By using this modelbuilder it is possible to create a full working COSTA model component by only implementing a very small number of routines.

The routines that are supported in the current version of the SP Model builder are:

- cta_model_create
- cta_model_free (not yet supported)
- cta_model_compute
- cta_model_setstate
- cta_model_getstate
- cta_model_axpymodel
- cta_model_axpystate
- cta_model_setforc (not yet supported)
- cta_model_getforc (not yet supported)
- cta_model_axpyforc
- cta_model_setparam (not yet supported)
- cta_model_getparam (not yet supported)
- cta_model_axpyparam (not yet supported)
- cta_model_getnoisecount
- cta_model_getnoisecovar

- cta_model_getobsvalues
- cta_model_getobsselect
- cta_model_addnoise

Not all methods are supported in the current release of the modelbuilder. The modelbuilder will support in the near future however.

Using the modelbuilder the model programmer only needs to implement a small number of subroutines. The modelbuilder will use these subroutines for implementing all methods. The subroutines that must be provided by the model programmer and their interface are given in the following sections.

11.3.1 Create a new model instance

This routine creates and initialises a new model instance.

`USR_CREATE(hinput, state, sbound, sparam, nnoise,`
time0, snamnoise, husrdata, ierr)

`IN`	`hinput`	Model configuration CTA_Tree of CTA_String
`OUT`	`state`	Model state (initialized to initial value Note this statevector must be created
`OUT`	`sbound`	State-vector for the offset on the forcings. CTA_NULL if not used Note this statevector must be created
`OUT`	`nnoise`	The number of noise parameters in model state is 0 in case of a deterministic model
`OUT`	`time0`	Time instance of the initial state state The time object is already allocated
`OUT`	`snamnoise`	Name of the substate containing the noise parameters The string object is already allocated
`OUT`	`husrdata`	Handle that can be used for storing instance specific data
`OUT`	`ierr`	Return flag CTA_OK if successful

```
   void usr_create(CTA_Handle *hinput,  CTA_State *state, CTA_State sbound,
                   CTA_State *sparam, int *nnoise, CTA_Time time0,
                   CTA_String *snamnoise, CTA_Handle *husrdata, int *ierr)
```

```
USR_CREATE(hinput, state, sbound, sparam, nnoise, time0,
           snamnoise, husrdata, ierr)
integer hinput, state, sbound, sparam, nnoise, time0
integer snamnoise, husrdata, ierr
```

11.3.2 Compute

This routine is computes several timesteps over a giving timespan.

`USR_COMPUTE(timespan,state, saxpyforc, baddnoise, sparam, husrdata, ierr)`

`IN`	`timespan`	Timespan to simulate
`IN/OUT`	`state`	State vector
`IN`	`saxpyforc`	Offset on models forcings
`IN`	`baddnoise`	flag (CTA_TRUE/CTA_FALSE) whether to add noise
`IN`	`sparam`	Model parameters
`IN/OUT`	`husrdata`	Instance specific data
`OUT`	`ierr`	Return flag CTA_OK if successful

```
void USR_COMPUTE(CTA_Time *timespan, CTA_State *state, CTA_State *saxpyforc,
                 int *baddnoise, CTA_State *sparam, CTA_HAndle *husrdata,
                 int *ierr)

USR_COMPUTE(timespan,state, saxpyforc, baddnoise, sparam, husrdata, ierr)
integer timespan,state, saxpyforc, baddnoise, sparam, husrdata, ierr
```

11.3.3 Covariance matrix of the noise parameters

This routine is responsible for returning the covariance matrix of the noise parameters.

`USR_COVAR(colsvar,nnoise, husrdata, ierr)`

`OUT`	`colsvar(nnoise)`	covariance of noise parameters array of noise Note the substates are already allocated
`IN`	`nnoise`	Number of noise parameters
`IN/OUT`	`husrdata`	Instance specific data
`OUT`	`ierr`	Return flag CTA_OK if successful

```
void usr_covar(CTA_State *colsvar, int *nnoise, CTA_Handle *husrdata, int *ierr)

USR_COVAR(colsvar, nnoise, husrdata, ierr)
integer nnoise, husrdata, ierr
integer colsvar(nnoise)
```

11.3.4 Model state to observations

This routine is responsible for the transformation of the state-vector to the observations.

`USR_OBS(state, hdescr, vval, husrdata, ierr)`

IN	state	state vector
IN	hdescr	Observation description of observations
OUT	vval	Model (state) values corresponding to observations in hdescr
IN/OUT	husrdata	Instance specific data
OUT	ierr	Return flag CTA_OK if successful

```
void usr_obs(CTA_State *state, CTA_ObsDescr *hdescr, CTA_Vector *vval,
             CTA_Handle *husrdata, int *ierr)

USR_OBS(state, hdescr, vval, husrdata, ierr)
integer state, hdescr, vval, husrdata, ierr
```

11.3.5 Observation selection

This routine is responsible for producing a selection criterion that will filter out all invalid observations. Invalid observations are observations for which the model cannot produce a corresponding value. For example observations that are outside the computational domain.

`USR_OBSSEL(state, ttime, hdescr, sselect, husrdata, ierr)`

IN	state	state vector
IN	ttime	timespan for selection
IN	hdescr	observation description of all available observations

OUT	sselect	The select criterion to filter out all invalid observations
IN/OUT	husrdata	Instance specific data
OUT	ierr	Return flag CTA_OK if successful

```
void usr_obssel(CTA_State *state, CTA_Time *ttime, CTA_ObsDescr *hdescr,
        CTA_String *sselect, CTA_Handle *husrdata, int* ierr)

USR_OBSSEL(state, ttime, hdescr, sselect, husrdata, ierr)
integer state, ttime, hdescr, sselect, husrdata, ierr
```

11.3.6 xml-configuration

The modelbuilder need to be configured in order to create a new model. This configuration specifies which functions are provided to implement the model.

The configuration has the following form (in xml)

```
<modelbuild_sp>
<functions>
   <!-- The functions that implement the model -->
   <create>my_create</create>
   <covariance>my_covar</covariance>
   <getobsvals>my_obs</getobsvals>
   <compute>my_compute</compute>
   <getobssel>my_getobssel</getobssel>
   <model>
   <!-- Everything overhere is passed through to the model (input argument hinput of th
   </model>
</functions>
</modelbuild_sp>
```

This configuration file is read into a COSTA-tree and is used as input argument for each instance that is created.

The names of the functions eg. my_compute, correspond to the name specified when administrating the function in COSTA using the `cta_func_create`.

Future versions of the modelbuilder will support dynamic linking to the user functions. When this is supported it will be possible to directly link the routines from the dynamic link library.

11.3.7 Examples

The modelbuilder is used for the models lorenz96, lorenz, and oscill in the COSTA model-directory. These models are a source of information concerning the use of this modelbuilder.

Chapter 12

The COSTA parallel modelbuilder

Remark: COSTA is incorporated in `OpenDA`, `/public/core/native/src/cta`.

Contributed by:	Nils van Velzen, CTA memo200606
Last update:	04-2014

12.1 Introduction

The COSTA environment makes a number of building blocks available for creating data assimilation and calibration systems. Combining and creating building new building blocks should be possible with a minimum of effort.

COSTA contains tools for rapidly creating COSTA model components. These tools are called modelbuilders. This document describes the parallel model builder. This modelbuilder will create a mode-parallel model from an arbitrary COSTA model.

This memo describes the first version of the parallel modelbuilder. Section 12.2 describes the form of parallelization implemented by the parallel modelbuilder. The usage of the modelbuilder and how to adjust your existing sequential data assimilation system is described in Section 12.3. The design of the parallel modelbuilder is described in Section 12.4. The modelbuilder is tested with a number of models and data assimilation methods. The results of these tests are presented in Section 12.5. This document describes the initial version. The testing and development stages yielded some ideas for future improvement of the modelbuilder. These ideas are presented in Section 12.6.

12.2 Mode-parallel

Data assimilation and model calibration algorithms need to perform a large number of model computations for a given timespan, called model propagations. These propagations are often very computational expensive and dominate the total time needed to do the assimilation or calibration run.

Not all propagations depend on each other, therefore it is possible to perform them in arbitrary order and in parallel. For example:

- The propagations of all ensemble members of an ensemble Kalman filter;
- The propagation of the L-matrix of the RRSQRT-filter and the 'central' state.
- Computation of the gradient of the model with respect to some model parameters using a finite difference approach.

The COSTA parallel modelbuilder makes it possible to perform the propagations of different model instances in parallel. Significantly decreasing the computational time for most data assimilation applications.

12.3 Using the COSTA modelbuilder

12.3.1 Adjusting the code

In order to use the parallel modelbuilder it is necessary to make a small extension to the code of an (existing) COSTA data assimilation system. After the initialization of the model the function `CTA_MODBUILD_PAR_CREATECLASS` must be called. From that point the parallel modelbuilder is initialized and the available processes are divided into a master process and several worker processes.

`CTA_MODBUILD_PAR_CREATECLASS(modelcls)`

`OUT`	`modelcls`	Class handle of the parallel modelbuilder

```
void CTA_Modbuild_par_CreateClass (CTA_ModelClass *modelcls)

CTA_MODBUILD_PAR_CREATECLASS (MODELCLS)
   INTEGER MODELCLS
```

Note that the worker processes will not execute any code after this function call. As a consequence, the modelbuilder must be initialized after the class initialization of the model. The parallel model class handle must be used for the creation of all models. The XML-configuration of the modelbuilder will realize the link with the existing simulation model. This is explained in Section 12.3.2.

The First lines of a typical application using the parallel modelbuilder are similar to the code in Table 12.1.

```
call cta_initialise(retval)
!
!    Initialise model (oscillation model)
!
call oscill_model_createfunc()
!
! Initialise parallel modelbuilder and start workers
!
call cta_modbuild_par_createclass(CTA_MODBUILD_PAR)
!
!    Process input and call method
!
   :
call cta_finalise(retval)
```

Table 12.1: *Main program of a data assimilation system using the parallel modelbuilder.*

12.3.2 input-file

The input of the modelbuilder is quite simple. The input of a simulation system using the parallel modelbuilder is given in Table 12.2. In the current version only two things need to be specified:

- name (tag) of the modelclass of the model,
- the name of the input-file of the model.

```
<modelbuild_par modelclass="modelbuild_sp">
   <model>oscill.xml</model>
</modelbuild_par>
```

Table 12.2: *The input-file of the parallel model-builder*

As we can see there is one limitation compared to arbitrary models. The model-input can only consist of a single string, in most cases the name of a file containing the model configuration. Most models can however be quickly adjusted, when necessary. COSTA provides a special function `CTA_Model_Util_InputTree` for handling the input and configuration of model instances. If a COSTA tree is given at model creation nothing is done. When however a string with the name of XML-configuration file is passed, the file is parsed and the corresponding configuration tree is created.

```
CTA_MODEL_UTIL_INPUTTREE(hinput, tinput, cleanup)
```

IN	`hinput`	Models configuration; a string of the configuration file or a COSTA tree
OUT	`tinput`	COSTA tree with model configuration. When `hinput` is a COSTA tree than `tinput` is equal to `hinput`.
OUT	`cleanup`	Flag `CTA_TRUE`/`CTA_FALSE`. When `tinput` is a filename, a COSTA tree is created and this tree must be cleaned/freed by the caller of this function.

```
int CTA_Model_Util_InputTree(CTA_Handle hinput, CTA_Tree *tinput,
                             int *cleanup)
```

```
CTA_MODEL_UTIL_INPUTTREE(HINPUT, TINPUT, CLEANUP)
   INTEGER HINPUT, TINPUT, CLEANUP
```

12.3.3 Axpy model between two model instances

The Axpy operation between two models is supported in the parallel model builder. There is a limitation. In order to perform the Axpy for models that are on different workers a copy of the model on the remote worker must be created. Therefore it is necessary that Export and Import methods of the model are implemented.

12.3.4 Random numbers

Stochastic models will use a random generator. When working with multiple processes we must be careful. There is a change that all processes will generate the same sequence of random numbers leading to undesired results.

The modelbuilder handles this issue. Therefore nothing needs to be done for models that use the COSTA random generator `CTA_RAND_N` or a random generator that is based on the `rand` function of C and did not set the random seed (`srand`).

Models that use a different random generator must be checked and optionally adjusted.

12.3.5 Running

The parallel modelbuilder is uses the MPI system for starting up the processes and handling the communication between these processes. The program `mpirun` or `mpiexec` must be used to start the application. It is far behind the scope of this document to describe all features of MPI, `mpirun` and `mpiexec`. We will only give some instructions on how to start up your program in a way that will work for most MPI distributions.

In order to startup your program `myfilter.exe` using 5 processes type on the command line:

```
% mpiexec -n 5 myfilter.exe
```

or

```
% mpirun -np 5 myfilter.exe
```

In this case we will have 1 master and 4 workers.

Some MPI distribution allow to start a sequential simulation without the use of `mpirun` or `mpiexec`.

```
% myfilter.exe
```

However this does not always work. If this does not work, start the sequential version of the application using:

```
% mpiexec -n 1 myfilter.exe
```

Note that the master process is idle for most assimilation methods when the workers are performing their computations. Therefore it is possible to start $n+1$ processes on n processors.

The model instances are equally distributed over the available workers. For that reason it is wise to select the number of model instances e.g. ensemble members as a multiple of the number of worker processes. When this is not done not all resources are optimally used. The propagation of 11 modes on 5 processors will take as much time as the propagation of 15 modes on the same number of processes for example.

12.4 Technical aspects of the modelbuilder

12.4.1 Master worker

The goal of the parallel modelbuilder is to provide parallel computing and decrease computational (wall) time of computations on clusters of workstations or advanced multiprocessor machines. The modelbuilder must be easy to use with no or minimum adjustments to existing models and assimilation method implementations.

COSTA models have no idea about the context they are used in. Therefore they have no clue on what methods in what sequence are called. Configuration files describing all algorithmic steps can be used to tackle this. This approach will result in an optimal performance and is used in Roest and Vollebregt (2002). The creation of a detailed configuration file it quite complicated and configuration files have to be written for all assimilation methods. To overcome this problem a master worker approach is chosen for the parallel modelbuilder.

In the master worker approach we use a single master process. In our case this process will run the assimilation algorithm like the sequential case. The remaining processes are worker processes executing all model component related computations. Depending on the number of model instances a worker holds one or more model instances.

The necessary information, the header variables of the call, will be send to the worker process when the master executes a method of a COSTA model. Whenever the method returns information, the master will wait for the worker to send the result back. Some optimization can be performed using non-blocking communication but this issue will be discussed in more detail in Section 12.6.7. The advantage of this approach is that the assimilation code does not need to be adjusted. All communication is handled by the parallel modelbuilder. The disadvantage is however that the parallelization will not be optimal.

12.4.2 Sequential bottleneck

The parallel modelbuilder will speedup most data assimilation systems but the improvement is limited by the sequential parts in the assimilation method. We will illustrate this using the ensemble Kalman filter.

A time step of the ensemble Kalman filter consists of the following steps:

1. Propagate all ensemble members
2. Assimilate observations and adjust state of all modes

The propagation part 1 is performed in parallel. In the most optimal situation it will take the time of a single mode propagation. The assimilation part 2 is performed by the master process. The usage of the modelbuilder will not improve the computational time of this part[1].

The wall time of the computations in the sequential run is approximately defined by

$$t_{wall} = nt_{prop} + t_{assim} \tag{12.1}$$

Where n denotes the number of members in the ensemble. Increasing the number of workers will only influence the wall time of the propagation. The lower bound on the computational time using the parallel modelbuilder is therefore

$$t_{wall} = t_{prop} + t_{assim} \tag{12.2}$$

The simulation results that are presented in Section 12.5 will illustrate this behavior.

[1]Due to some communication with the model for performing interpolation and setting of states it is likely that this step will be more expensive than in the sequential case

12.4.3 Sequential runs

A data assimilation system that uses the parallel modelbuilder should be equally efficient as the sequential implementation when running on one process. The parallel modelbuilder recognizes when it is initialized in a sequential run. In that case all methods of the model are directly connected to the parallel modelbuilder.

12.4.4 Communications

The master will send information to the worker when a method of the model is called and will wait for the results. In this section some insight is given on the actual implementation.

Local administration

For each model instance the master holds a limited administration currently only containing the rank of the worker that holds the model instance and the (integer) handle of the model instance at the worker.

Packing of data

In order to send information between the processes it is possible to pack and unpack COSTA objects like vectors state vectors and observation description instances. When an object is packed all information is copied into a continuous block of memory. The unpack operation will reconstruct the object from the packed information.

The COSTA pack component is used to hold packed data. Multiple components can be packed and unpacked into a single pack component. The export and import methods of components will perform the actual packing and unpacking.

Communication sequence

The parallel modelbuilder will perform the following actions when the worker executes a method of the model.

The master performs the following steps:

1. Send the name of the action including the local model handle to the worker handling the model instance,
2. (optional) pack all data; input arguments of method,
3. (optional) send packed input arguments to method,

4. (optional) receive packed results from worker.
5. (optional) unpack results from worker.

The worker performs the following steps:

1. receive the name of the action and local model handle
2. (optional) Receive packed input arguments,
3. execute method,
4. (optional) pack returned data; output arguments of method,
5. (optional) Send packed output arguments to method.

Some of the steps are marked as optional. It depends on the method that is called whether these steps are performed or not.

12.5 Tests and performance

12.5.1 Small test-models

The parallel modelbuilder is tested using two existing COSTA test models;

1. the heat model with the COSTA RRSQRT filter,
2. the oscillation model with the COSTA ensemble Kalman filter.

The purpose of these tests was to find errors in the modelbuilder. Since these models are very small we don not expect a lot of improvement in performance. These tests are therefore not run on a cluster of computers or a large multiprocessor machine.

Figure 12.1 gives the simulation results of a parallel and sequential run of the Ensemble Kalman filter for the oscillation model. The small differences in the results are due to different random seeds.

12.5.2 Lotos-Euros

Lotos-Euros is an operational used simulation model for air pollution in Europe. The model computations are computational expensive compared to the small models like the oscillation and heat model. The COSTA model component of this model was already available and therefore we have selected this model to look at the performance of the parallel model builder.

The tests have been performed on two different machines.

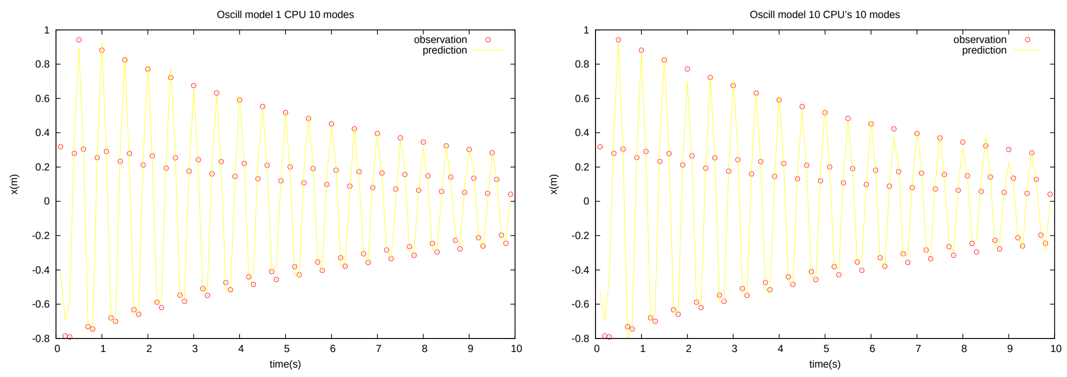

Figure 12.1: *Results of the Ensemble Kalman filter with 10 modes for the oscillation model. The left graph presents the results of a sequential run and the right graph the results of a parallel run with 10 workers. Differences are the result of different random seeds.*

- Laptop with Intel centrino duo, 1.66GHz CPU,
- Aster, an SGI Altix 3700 system with 416 Intel Itanium 2, 1.3GHz CPU's

The tests on the Intel centrino duo processor will illustrate that parallel computing is also useful on a single machine with a dual core processor. The simulation results are given in Table 12.3.

Gridsize	1 process	3 processes	Speedup
30x30x4	43.9 (s)	29.9 (s)	1.5
60x60x4	234.7 (s)	164.8 (s)	1.4

Table 12.3: Sequential and parallel run of the Lotos Euros model on a dual core processor. The run is an Ensemble Kalman filter with 24 modes for a simulation period of 2 hours.

The same simulation is also performed on the Aster on a number of CPU's varying from 1 to 12. The results are presented in Figure 12.2 and Figure 12.3. The speedup of the small model is much better than of the big model. We have not yet investigated this behavior. A possibility is that the sending of the states to worker is nonblocking for the small model as a result of an optimization in the MPI implementation and blocking for the large model.

12.5.3 Observation description

The default observations handling in COSTA is currently based on an SQLITE-database. Testing with the Lotos-Euros model revealed a performance issue. The time needed for reading from the database is not constant and requires a lot of wall-time, degenerating the performance of the parallel implementation.

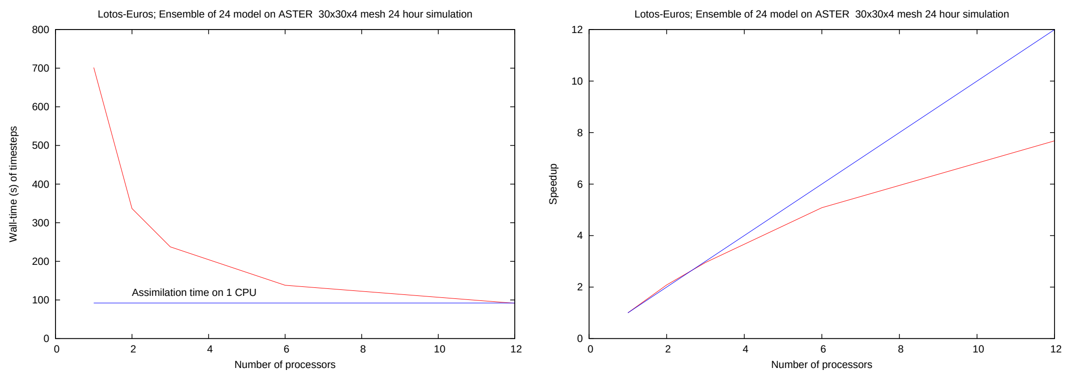

Figure 12.2: *Computational time and speedup. Lotos Euros simulation of 24 hours grid size 30x30x4 on ASTER. The ensemble Kalman filter used 24 model instances (23 members for covariance approximation and 1 background run)*

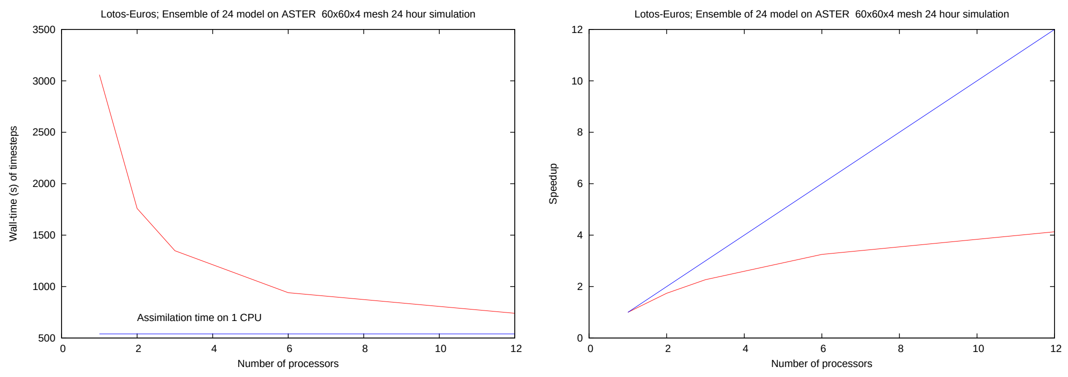

Figure 12.3: *Computational time and speedup. Lotos Euros simulation of 24 hours grid size 60x60x4 on ASTER. The ensemble Kalman filter used 24 model instances (23 members for covariance approximation and 1 background run)*

The parallel modelbuilder will will remember the data from the last send observation description. This will save some time because the information only needs to be read from the database once and not for all workers. This performance issue needs to be investigated in more detail in the future.

12.6 Future improvements

12.6.1 MPI initialization

The current implementation uses the `MPI_COMM_WORD` communicator and initializes MPI. For creating combinations with models that use MPI by them selves, this does not work. The initialization of the parallel model builder must be extended in order to create separate communication groups for the COSTA master and worker processes and the used model implementation.

12.6.2 Model state on master

The sequential bottleneck of the getstate method can be overcome by holding a model state of each model instance at the master. The value of this state is set using a nonblocking communication after every propagation.

This option can be switched on in the configuration file.

12.6.3 States on workers

The worker processes are now limited to holding model instances. The implementation of the workers is general and not limited to handling models. An extension to the parallel modelbuilder is the possibility to remotely hold state-vectors. This has two advantages:

1. part of the linear algebra computations with state-vectors in the assimilation part of the data assimilation method can be performed in parallel resulting in a better performance,
2. when the computations are performed on a non-shared memory machine the memory consumption is better distributed over the various machines and larger models and more model instances can be handled.

12.6.4 Constant obsselect

When the obsselect method results a constant selection criterion during the whole simulation. It can be specified in the input, saving communications.

12.6.5 Column selection for observation description

When the model must provide values that correspond to the observation description component, the whole observation description component is packed and send. In some cases not all data is needed by the model to perform the interpolation. Some of the data is only relevant for post processing etc. The column selection option in the configuration file of the parallel modelbuilder will specify what columns of the observation description component need to be send to the model.

12.6.6 Mode on all workers

In some situations like RRSQRT we have a central mode that is used in a large number of computations. A lot of communication can be saved when the value of this central state resides on all workers. This option is specified in the input-file of the modelbuilder.

12.6.7 Nonblocking communication

All receiving of data is currently blocking. This means that the Master must wait until the worker has send the requested data. MPI offers the possibility of a nonblocking receive. In the case of a non blocking receive the master will indicate at what position in memory it expects the data to put and continues. At the point the Master is actually going to use the data it has to wait until it is received.

The usage of the non-blocking receive can eliminate some of the sequential parts of the filter algorithm. For example; the assimilation method wants to have a copy of the states of all model instances using the following code:

```
do imode = 1,nmode
   call cta_model_getstate(hmodel(imode),sl(imode),ierr)
enddo
```

In the current implementation this procedure is sequential. Because no new state is requested before a state is received and unpacked.

Using non-blocking receive (what will probably need an extension of the state-vector) this operation can be performed parallel.

Chapter 13

Parallel Computing in COSTA

Remark: COSTA is incorporated in `OpenDA`, `/public/core/native/src/cta`.

Contributed by:	Nils van Velzen, CTA memo 200801
Last update:	04-2014

13.1 Introduction

Simulation models can be very large in terms of memory usage and computational requirements to perform a simulation. For these large models it is sometimes necessary to use parallel computing in order to be able to perform a simulation in a reasonable time. Most data assimilation and model calibration methods will perform a large number of model runs. Therefore increasing the computational time significantly compared to a normal simulation. Parallel computing is a vital part for large simulation models and for that reason COSTA supports the usage of parallel simulation models as well the auto-parallelization of non-parallel models. In this report we will give a description of the parallel computing capabilities of COSTA.

13.2 Almost invisible for user

COSTA is an environment that must be easy to use. The parallel computing facilities are therefore set up in such a way that they are easy to use. To simplify usage for users who do not need parallel computing it is developed in such a way that is is completely hidden when it is not needed.

At the installation it is possible to install COSTA without parallel support. This simplifies the installation since no third party MPI library need to be installed on the system. There is only a single additional call to an initialization function necessary in order to use the parallel functionality and a minimum of additional configuration needs to be added to the configuration files compared to sequential runs.

13.3 Kinds of parallelism

There are various methods to incorporate parallel computing in a simulation model. Two popular methods are:

- a single process with multiple threads. Within a single executable multiple threads are started to execute parts of the code in parallel. These kind of parallel computing can by implemented by using e.g. OpenMP,
- multiple processes that together perform the computations. The data is distributed over the various processes and the processes cannot directly access each other data. Information is pased between the processes by sending messages to each other. The software libraries MPI and PVM are popular for developing these kind of programs.

Models that are parallelized using threads are not different than normal sequential models as far as COSTA concerns. These models can therefore be transformed into a COSTA model component as any normal sequential model.

Special functionality is added to COSTA in order to be able to link to models that are parallelized using the second concept. The model computations will take place in different executables as the data assimilation computations and the various methods of the model are realized by sending messages between the data assimilation method and the model. However all communication is hidden behind the COSTA model interface. The data assimilation code that is used for a sequential model is therefore exactly the same as for parallel models.

The support for parallel computing in COSTA is based on MPI since it is at this time the most widely supported and used library for creating parallel simulation models. In theory it is also possible to create hybrid parallel applications that are based both on PVM and MPI but we have no experience with this.

There are various approaches to introduce parallelism in a model. COSTA supports the following two:

1. Master worker; One process called the master is special. This process solves the problem by giving the workers tasks.
2. Worker worker; The problem is split up into peaces and all processes work together in order to solve part of the problem. This kind of parallelism is used e.g. when:
    - a large computational area is split up into parts.
    - two different models are combined into a larger model. Like a runoff model with a river model.

Figure 13.1 illustrates the coupling between a data assimilation method and a parallel model. The main difference between the coupling between a Master-Worker and Worker-Worker process is the communication between the data assimilation method and the model. In the

Master-Worker situation there is only communication between the data assimilation method and the master process of the model. In case of a Worker-Worker model, the data assimilation method communicates with all worker processes.

13.4 How does it work

A parallel COSTA application consists of a number of processes (executables that are started). One of the executables implements the data assimilation method and it has the task of a master process. The other executables are worker processes and used for performing the model computations.

When a method is used from the data assimilation method then COSTA will send a message containing all necessary information to the worker processes that implement the model.

In the case the model itself is parallelized using the master-worker principle, the message is only send to the master process of the model. Models that are parallel using the worker-worker principle are handled like a concatenation of COSTA models. All messages are send to all the worker processes and the overall state of the model is the concatenation of the states of all the worker processes. The concatenation is handled by COSTA and invisible.

13.5 Process groups in MPI

Parallel models that are linked to COSTA will run in a larger group of processes than they are original developed for. Fortunately MPI provides the concept of process groups and communicators.

COSTA defines three communicators that can be used for communication between the various processes:

- CTA_COMM_WORLD; The communicator for all processes in the COSTA universe.
- CTA_COMM_MYWORLD; The communication group of the parallel model. This communicator should be used for all existing communication calls in the model.
- CTA_COMM_MASTER_WORKER; The communication group consisting of master process (data assimilation method) and all worker processes (processes that are used by the model) the master directly communicates with.

The communication groups are created by the function `CTA_Par_CreateGroups` at startup of the processes. This routine needs configuration that is specified in an XML-input file. For all models it must be specified how many processes are needed for a single model instance (`nproc`), how many of these groups we want to create (`ntimes`), and the kind of parallel computing is used inside the model "Master-Worker" or "Worker-Worker" (`parallel_type`).

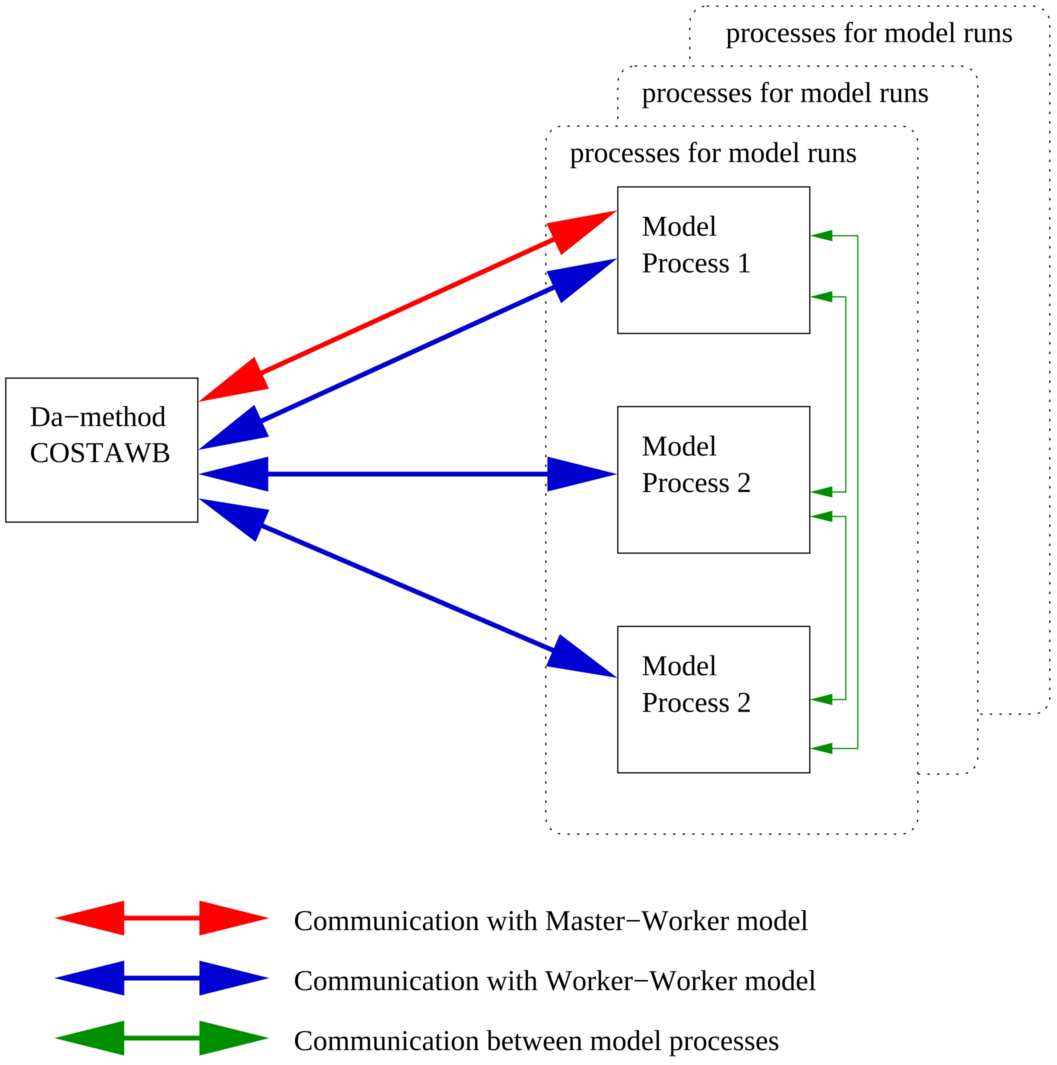

Figure 13.1: *Example of a coupling with COSTA and a parallel model. When the model is parallel according to the Master-Worker concept, there will only be communication between the data assimilation method and the Master process of the model. When the model is parallel according to the Worker-Worker concept, it is necessary to communicate between the data assimilation method and all worker processes. In this example we see that it is possible to create multiple groups of model processes each holding there own model instances*

There are two ways to specify this in the input file. In this first form the parallel attributes are added to the definition of the model class. For example:

```
  <CTA_MODELCLASS id="modelclass"
                  implements="pollute2d"
                  name="CTA_MODBUILD_SP"
                  nproc="2"
                  parallel_type="Worker-Worker"
                  ntimes="*"/>
```

In this example the 2d pollution model is parallelized using the Worker-Worker principle. This model uses 2 processes. The option `ntimes="*"` indicates that as many processes will be used as available to distribute the available model instances.

For example: when 5 processes are started then one process will run the data assimilation method and there will be two groups of two processes both handing half of the created model instances.

An other option is to specify the information separate from the model class definition. This can be useful for the configuration of worker processes that do not need to define the model classes of other models. For example

```
<parallel>
   <process_groups>
       <group name="group_of_2", nproc="2", use_for_model="pollute2d",
              parallel_type="Worker-Worker" ntimes="*">
       <group name="group_of_3", nproc="3", use_for_model="rainfall",
              parallel_type="Master-Worker" ntimes="2">
   </process_groups>
</parallel>
      :
      :
<CTA_MODELCLASS id="modelclass"
                implements="pollute2d"
                name="CTA_MODBUILD_SP"/>
```

Note that the `implements` tag must correspond to the `use_for_model` in the group specification. It is allowed to mix both ways of specification in a single input file.

13.6 Starting up parallel runs

The COSTA workbench program `costawb` can be used both for parallel as sequential runs. The only difference is the need of an additional argument `-p` to indicate that a parallel run is started. Depending on the MPI distribution the processes must be started using `mpiexec` or `mpirun`. Note that the exact usage of these startup programs differs between the various MPI distributions. The examples we give here work for the Lam-MPI distribution. A parallel run can be started with the command:

```
mpiexec -np 3 costawb -p input.xml
```

This will start 3 processes. Two are available for the model and one for the data assimilation method itself.

The 2D pollution model is one of the test models in COSTA. There are two parallel versions of the model available. A Worker-Worker and a Master-Worker version.

The Worker-Worker version of the model can be started using

```
mpiexec -np 3 costawb -p ens_pollute2d_ww.xml
```

since all processes are the same. The Master-Worker version is however different. The Worker processes of the model are different executables. The processes can be started using the command:

```
mpiexec -np 2 costawb -p ens_pollute2d_mw.xml :\
        -np 1 pollute2d_worker ens_pollute2d_mw.xml
```

This will start 3 processes. The first two `costawb` handle the data assimilation method and the Master process of the model and `pollute2d_worker` is the worker process of the model.

13.7 Creating a parallel COSTA model

13.7.1 Master-Worker models

The Master process implements the COSTA model interface just like a sequential model in case of a Master-Worker parallelization of the model. The worker processes only need to call the function `CTA_Par_CreateGroups` at startup to create the process groups and communicators. The worker processes have in general their own executable. For example the code of the executable of the worker processes of the pollution model is give by:

```
program pollute2d_worker
use pollute2d_params, only:pollute2d_params_init
use pollute2d,        only: create_state_vector
implicit none
include 'cta_f77.inc'
```

```
integer ::state   !State vector
integer ::ierr    !Error code
integer ::hfile   !Name of configuration file
integer ::htree   !Costa tree representation of input file
character(len=1024) ::inpfile

   !Get name of input file from command line
   if (iargc()/=1) then
      print *,'Program must have 1 argument'
      call exit(-1)
   endif
   call getarg(1, inpfile)

   call CTA_Initialise(ierr)

   ! Read configuration file (contains the procress-group information)
   call CTA_String_Create(hfile, ierr)
   call CTA_String_Set(hfile, inpfile, ierr)
   call CTA_XML_Read(hfile,htree, ierr)
   if (ierr/=CTA_OK) then
      print *, 'Error opening or parsing file ',inpfile
      call exit(1)
   endif

   ! Create process groups but do not start model builder
   call CTA_Par_CreateGroups(htree,CTA_FALSE, ierr)

   ! Perform initializations
   call pollute2d_params_init

   ! Create state vector
   call create_state_vector(1, 0, state, ierr)

   ! Wait for tasks of the master
   do
      call pollute2d_compute(CTA_NULL,state, CTA_NULL, CTA_FALSE , &
                             CTA_NULL, CTA_NULL, ierr, .false.)
   enddo

end program
```

13.7.2 Worker-Worker models

The model component is programmed exactly in the same way as a sequential model. When one of the methods of the model is used from the data assimilation method it will result in a call to the corresponding method at all worker processes. The communicator `CTA_COMM_MYWORLD`) can be used by the model to communicate to the other processes that implement the model.

References

Byrd, R., Nocedal, J., and Schnabel, R. (1994). Representations of quasi-newton matrices and their use in limited memory methods. *Mathematical Programming*, 63(4):129–156.

Daley, R. (1991). *Atmospheric data analysis*. Cambridge University Press.

Drécourt, J.-P., Madsen, H., and Rosbjerg, D. (2006). Bias aware Kalman filters: Comparison and improvements. *Advances in Water Resources*, 29(5):707–718.

Evensen, G. (1994). Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics. *Journal of Geophysical Research*, 99(C5):10,143–10,162.

Heemink, A., , Verlaan, M., and Segers, A. (2001). Variance reduced ensemble kalman filtering. *Monthly Weather Review*, 129:1718–1728.

Lorenz, E. N. (1963). Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20(2):130–141.

Lorenz, E. N. and Emanuel, K. A. (1998). Optimal sites for supplementary weather observations: Simulation with a small model. *Journal of the Atmospheric Sciences*, 55:399–414.

Nelder, J. and Mead, R. (1965). A simplex method for function minimization. *The Computer Journal*, 7(4):308–313.

Nocedal, J. (1980). Updating quasi-newton matriced with limited storage. *Mathematics of Computation*, 35(151):773–782.

Roest, M. and Vollebregt, E. (2002). Parallel Kalman filtering for a shallow water flow model. In Wilders, P., Ecer, A., Periaux, J., Satofuka, N., and Fox, P., editors, *Parallel Computational Fluid Dynamics - Practice and Theory*, pages 301–308.

Talagrand, O. (1997). Assimilation of observations, an introdction. *Journal of the Meteorological Society of Japan*, 75:191–209.

Tippett, M., Anderson, J., Bishop, C., Hamill, T., and Whitaker, J. (2003). Ensemble square root filters. *Monthly Weather Review*, 131:1485–1490.

Verlaan, M. and Heemink, A. (1997). Tidal flow forecasting using reduced rank square root filters. *Stochastic Hydrology and Hydraulics*, 11:349–368.

Appendix A

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “**publisher**” means any person or entity that distributes copies of the Document to the public.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as

4. MODIFICATIONS

or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time