Princetonlaan 6
3584 CB Utrecht
P.O. Box 80015
3508 TA Utrecht
The Netherlands

www.tno.nl

T 31 88 866 4256
F 31 88 866 4475

TNO Report

TNO 2016 R10899Report

LOTOS-EUROS v2.0 User Guide

Date July 2016

Author(s) Arjo Segers
Astrid Manders
Sander Jonkers

No. of pages 45 (incl. appendices)

Contents

1 Introduction

1.1 What can be found in this guide?

This is the User Guide of the LOTOS-EUROS model. The purpose of this guide is to provide information on how to:

- obtain a copy of the source code;
- compile and setup a simulation;
- run the model
- add new parts to the code

For information about the physical processes and parameterization used in the model we refer to Manders et al. (2016).

Not all parts of the user guide are relevant for a new user. Once the LOTOS-EUROS model has been installed on the user's system, Chapter 4.2 and Chapter 5 should contain all the relevant information to get started. For a new installation however all paths to data directories, output directories, libraries etc. have to be set for which more in-depth knowledge is needed, like the use of macro's, libraries and the structure of meteorological fields. Chapters 9, 11 and 12 are useful when a user would like to be more flexible in terms of grid, resolution, tracers and ouput, without going into details of the model.

1.2 Raw documentation

The original text of this User Guide is written in LaTeX format. The files are included in the LOTOS-EUROS distribution:

```
lotos-euros/v2.0/doc/user-guide/tex
```

A pdf version can be created with:

```
make pdf
```

Similarly, a browsable version can be created with:

```
make html
```

Use 'make help' to see options and other targets.

2 Websites

2.1 Public model website

The main source of information on LOTOS-EUROS is the public website:

www.lotos-euros.nl

The website contains general information, links to the operational forecasts, documents, and an overview of publications for which the model was used.

2.2 SharePoint site

Documents and data sets specific for the open-source version can be found on the SharePoint site (username/password requested):

https://ecity.tno.nl/sites/OpenLE/

3 Version control system

The open-source version of LOTOS-EUROS is not available in an open version control system. Instead, package files are made available to users through the OpenLE SharePoint site (see section 2.2).

Users are encouraged to upload their contributions to the model to the sharepoint site. The development team of LOTS-EUROS will then consider integration within new releases.

4 Source files

4.1 Identification of a model version

Each model version is given a three-leveled key, for example ”v2.0.000”. Here, the first 2 numbers identify the base version number (”v2.0”) while the third one is a patch number (”000”). New base versions are usually released before the beginning of a new year. Throughout the year, patches might be released to fix a bug or to add new functionality to the current base version.

If you are a new user it is advised to simply use the latest version. As explained in the next section, list the directory ’base’ to see what is the latest patch in your copy.

4.2 Directory structure

The LOTOS-EUROS model is shipped in the following directory structure:

```
lotos-euros/v2.0/            # Version directory
             README      # First aid information.
             doc/        # Documentation specific to this version
             base/       # Base sources
                  000/   # Patch directories
                  001/   #
                    :    #
             proj/       # Modifications to the base source
             tools/      # scripts for meteo preprocessing etc.
```

At the level of the patch number (000 etc), a number of sub-directories is present with prescribed names, that help to keep Fortran files, scripts, and other data clearly separated from each other:

```
base/000/src/       # Fortran sources
base/000/bin/       # scripts
base/000/rc/        # configuration files
base/000/data/      # data tables
          :
proj/testchem/000/src/
proj/testchem/000/bin/
                  :
```

During the setup of a run, a copy of the source files is created in a temporary build directory, and then compiled over there. The source directories are therefore not polluted by object and module files.

4.3 Patch directories

A patch is an official update of the base, for example with a bug fix or with a new feature that is supposed to become part of a new release. The code of a patch version is stored in the ’base’ directory:

```
base/000/     # initial base version
base/001/     # first update
base/002/     # second update
        :
```

The complete code of a base+patch is stored in the patch directory.

In the rest of this document we will simply refer to a 'base+patch' code as a 'base' version. If not specified explicitly, the initial patch number is '000'.

4.4 The concept of source projects

A base directory like 'base/000' contains a frozen version of the model. A base version should run without any problem.

In addition to a base, a project could be defined as a modification of files in the base source. A project could also include new files that are not part of the base yet. Typically, projects are defined to:

- implement a non-standard feature that is very specific to an infrequently used application.
- create special output required for some applications.
- test new module features.

For example, a source code could be combined from the following directories:

```
base/000/src/             # base+patch
proj/testchem/000/src/    # test code
proj/myoutput/000/src/    # user specific
```

In here, the 'base/000/src' directory contains a complete model source, from which some files are replaced by those from the 'proj/testchem/000' directory, and in addition by some from the proj/myoutput/000' directory. The order is important: the latest version of a file that is copied will be the one that is actually compiled.

A list of projects to be used should be specified in the run configuration file (explained in section 5.4). For this example it will look like:

```
my.source.dirs       :  base/000 \
                        testchem/000 \
                        proj/myoutput/000
```

Although not necessary, it is good practice to include the patch number as a sub-directory of a project. In this way it is immediately clear to which patch this project is an extension.

5 Running LOTOS-EUROS

This chapter describes how to configure and start a run with the LOTOS-EUROS model. In section 5.1 a quick start is presented; from section 5.2 onwards the steps are explained in more detail.

5.1 Quick start

LOTOS-EUROS can be run by taking the following steps. This is just to give a first impression or a quick reminder; for details, take a look at the sections that follow.

1. Go to the required version directory:

   ```
   cd lotos-euros/v2.0
   ```

2. Decide which patch number should be used, for example '000'. Template settings for this patch are available in:

   ```
   base/000/rc/lotos-euros-v2.0-000.rc
   ```

   This is the main rc file and once the version is properly installed often the only file that has to be adapted. Browse through it to see if the settings (e.g. run identification, time, domain, species, emissions) for your run are ok. Eventually create a copy and modify for a specific run.

3. Setup a run-directory and compile an executable using the setup script:

   ```
   ./base/000/bin/setup_le base/000/rc/lotos-euros-v2.0-000.rc
   ```

4. Follow the instructions written by the setup script.
   Probably, the instructions tell you to go to the displayed run directory and start the submit script:

   ```
   cd <rundir>
   ./submit_le lotos-euros.x lotos-euros.rc
   ```

   Alternatively: supply the '--submit' option to the 'setup_le' script to submit automatically after the setup is finished. The 'submit_le' script accepts options to let the job run in the background or in a queue system; see section 5.12 for details.

5.2 Version directory

The best place to setup and start the model is from what we will call the '*version directory*' directory. Change to that directory before following the next steps:

```
cd lotos-euros/v2.0
```

5.3 Run scripts

The scripts used to setup, start, and submit a LOTOS-EUROS run are placed in:

```
base/000/bin/
```

Two groups of scripts can be distinguished:

- the 'pycasso' scripts (PYthon Compile And Setup Scripts Organizer) used to compile and setup a run;
- the 'submit_le' scripts used to submit a run to for example a queue system.

The 'setup_le' script is the main script for each run. It is convenient to link the setup script to the version directory because the script is called frequently:

```
ln -s base/000/bin/setup_le setup_le
```

If the scripting changed in higher patch version, it might be necessary to let this link point to the latest version. The scripts that are used are always those that are in the same directory as setup script that will be called.

5.4 The rc-file

The configuration of a model run is done through a text file called the 'rc'-file. The abbreviation 'rc' comes from 'resource', or specifically from the Unix 'X' resource file on which the format of the file is based. However, also 'run configuration' is a suitable expansion. Several 'rc' files are available to keep the main .rc file as short as possible.

- the main .rc file, 'lotos-euros'-'v2.0'-'000'.rc in which runid, specification of patches and eventually project code, time, domain, emissions boundary conditions and species are set. It refers to the following other .rc files:
- the 'lotos-euros'-regions.rc in which several model domains are predefined
- the 'lotos-euros'-landuse.rc in which settings and filenames for the land use files are defined
- the 'lotos-euros'-meteo.rc in which paths and settings for input meteorology are defined
- the 'lotos-euros'-emissions.rc in which paths and settings for antropogenic emissions are defined
- the 'lotos-euros'-bound.rc in which paths and settings for several sets of boundary conditions are given
- the 'lotos-euros'-output.rc in which one can specify the species and fields that have to be put out.
- the 'lotos-euros'-expert.rc in which macros and supported species are defined. Should only be modified by experienced users
- the 'lotos-euros'-forecast.rc determines settings for forecasts
- the machine .rc file refering to system-specific settings like compiler and libraries
- the lotos-euros'-'v2.0'-'000'-anywhere.rc is a main rc file with default settings for running outside the European domain. It refers to different land use and emissions sets.

These files will be explained in more detail below.

5.4.1 Copy from template

Copy a template rc-file and give it a suitable name:

```
cp base/000/rc/lotos-euros-v2.0-000.rc test-lotos-euros.rc
```

Modify it using a text editor, or at least browse through it to see if the settings for your run are ok. It should be self-explanatory with sections for time, grid, species, chemistry, emissions, boundary conditions, landuse, output and forecast (in that order) and options to run in parallel (OpenMP, 8 threads maximum), use of restart files (recommended). Below, the format is explained and some .rc files are treated in more detail. The others should be self-explanatory when opened in an editor.

5.4.2 *Format of the rc-file*

In this section the ideas and conventions of the .rc file are explained. A simple example of a part in the rc-file could be:

```
! name of input file:
le.input   :   /data/a.txt
```

This assigns the value ”/data/a.txt” to the key ”le.input”. Tools are available for the run scripts and the model to read values from an rc-file given the keys. The basic format rules for the rc-file are:

- Keys and values are separated by ’:’ .
- Empty lines are ignored.
- Comment lines start with ’!’ and are ignored as well.

More advanced usage is possible, for example expansion of keys or environment variables, and conditional setting using if-statements. For a description of the features use:

```
base/000/bin/rcget --doc
```

The best way to learn about all configuration options is to browse through the template rc-file. The added comments explain the use of the various keys and the values they can take.

5.4.3 *The machine-specific rc-file*

An import setting in the main rc-file is the selection of the ’machine’ specific rc-file. This file contains all settings specific for the computer on which the model is running, e.g. compiler settings, library locations, data locations, etc. This file is for example:

```
base/000/rc/machine-tno-hpc.rc
```

For each institute/machine combination, a machine-specific rc-file should be present. If it is not, create a new one. To load the settings in the rc-file, select the appropriate machine rc-file:

```
my.machine.rc  :  machine-tno-hpc.rc
```

This variable is used elsewhere in the rc file to include the settings from the correct directory:

```
! include settings:
\hl{#include ${my.pycasso.dir}/rc/${my.machine.rc}}
```

See chapter 6 for information about the external libraries.

5.4.4 *The compiler rc-file*

The machine-specific rc-file includes a file with compiler-specific settings:

```
! compiler specific settings:
#include ${my.pycasso.dir}/rc/compiler-gcc-v5.3.0.rc
```

For each compiler suite used to compile the model, a compiler-specific file should be present.

5.4.5 The expert rc-file

Many settings that have to do with the setup and installation of the model have been hidden for the users by collecting them in the 'expert' rc-file, which is included into the main rc file:

```
! include expert settings to build source code
#include ${my.pycasso.dir}/rc/lotos-euros-expert.rc
```

Don't touch this file unless you know what you are doing.

5.5 Setup a run

Go to the 'lotos-euros'/'v2.0' directory. Call the script with the rc file as argument to setup a run-directory and compile an executable:

```
./setup_le test-lotos-euros.rc
```

To see the extra options that are accepted by the setup script, use:

```
./setup_le --help
```

Note that all 'long' options like '--help' usually have a 'short' version too, in this case '-h' .

A useful option is '--new' or '-n', which first removes the existing build directory followed by creating a completely new one. In case you receive strange errors from the compiler, that might have to do with messing up old and new objects, so try whether this option solves the problems.

Another commonly used option is '--jobs=1' or '-j 1' which ensures that source files are compiled one by one. Without this limitation, the 'make' program will try to compile as much files as possible simultaneously (if they can be compiled independently). Although the latter strategy is obviously much faster, it may mess up the messages printed to the screen.

5.6 The run directory

The first step taken by the 'setup_le' script is to create a run directory on a location specified in the rc-file. Typically the run-directory will be located on a scratch space, since a lot of temporary files are created while running that do not have to be backed-up. A typical content of the the run directory is:

```
<rundir>/build/     # source code, object files
<rundir>/run/       # executable, rc-files, submit script, log files
<rundir>/output/    # output files
<rundir>/restart/   # restart files}
```

5.7 Preprocessing of the rc-file

The next step taken by the 'setup_le' script is the preprocessing of the rc-file. The preprocessed file is put in the directory <rundir>/run/. Preprocessing the rc-file includes evaluation of environment variables and other features to make configuration easier. For an overview of all features, use:

```
base/lotos-euros/bin/rcget --doc
```

5.8 Collection of a source code

The next step is the collection of a source code in a build directory. The build directory is usually located on a temporary scratch disk, the exact location is specified in the rc-file. The sub-directory where the code is collected includes the name of the compiler and the choices for the compiler flags (see section 5.11 for the compiler flag settings). This is done to ensure that an executable is compiled with the same flags applied for all source files. An example of a sub-directory name:

```
<rundir>/build_optim-none_check-all/src/
```

The concept of source projects is explained in section 4.4. In summary, the first collected source files are those in the 'base/xxx' directory. Subsequently, the files from the base version are overwritten by patches or project specific versions from 'proj/' directories according to the specification in the main .rc file.

For example, the rc-file might contain the following setting for the list of source directories to be included:

```
my.source.dirs       :  base/000 \
                        proj/testchem/000 \
                        proj/myoutput/000
```

In this example, the LOTOS-EUROS source is formed from:

1. the files in 'base/000/src' ...
2. ... or those from 'proj/testchem/000/src/' ...
3. ... or those from 'proj/myoutput/000/src/'.

The test files in 'proj/testchem/000/src/' will not affect the 'official' versions of the code. Thus, if you want to change something in the code, create a new project directory and implement your changes there.

5.9 Generation of source files

Some source code files are generated by the scripts using settings in the rcfile and data tables. These are:

- source files with tracer definitions and chemistry codes (see section 11);
- preprocessing macro include files (section 7).

Default versions of the generated source files are included in the base source. Editing won't have any effect; instead change the rcfile settings or the tables that were used to generate them.

5.10 Creation of the Makefiles

The 'pycasso' scripting automatically creates the Makefiles using the 'makedepf90' program. If this program is not available, instructions are displayed.

5.11 Compilation

The final step performed by the 'setup_le' script is the compilation of the executable.

An important configuration choice for compiling is setting the compiler flags. By default, the executable is compiled with the 'fast' flags to have a run-time as low as possible. For testing and debugging it is however useful to enable the 'check' flags, which could for example trap out-of-array-bound problems and floating-point-exceptions (division by zero etc). *After changing the code, first run with checks enabled!*.

The flags to be applied can be set in the pycasso rc file by a list of keywords. The default setting for compilation with the fast flags is:

```
my.build.configure.flags : optim-fast
```

For testing and debugging purposes, the 'check' flags can be turned on with:

```
my.build.configure.flags : optim-none check-all
```

The actual flags assigned to these keywords are set in the compiler rc-file described in section 5.4.4.

5.12 Submit script

The LOTOS-EUROS executable should be started using the submit script.

First switch to the run directory. Its location can be found in the message displayed at the end of the setup by the 'setup_le' script, for example:

```
cd /scratch/yourname/PROJECT/test1/run
```

Next, execute the script with the executable and the rc file as arguments (see as well section 5.4.4):

```
./submit_le lotos-euros.x lotos-euros.rc
```

The submit script has some extra options. For example, to let the model run in the background (so that you can log out without killing the run), use:

```
./submit_le lotos-euros.x lotos-euros.rc --background
```

Similarly, use '--queue' to submit to a queue system. To see all options, use:

```
./submit_le --help
```

When a simulation is run in the background, an output file which contains a summary of the settings, info about the simulation, warnings and error messages will be available in the

```
<rundir>/run/
```

directory

5.13 Setup and submit in a single step

To setup and submit in a single step (so without the need to go to the run directory and call 'submit_le'), just add the '-s' or '--submit' option to the setup script:

```
./setup_le test-lotos-euros.rc --submit
```

The setup script also accepts the '--background' and '--queue' options that are then passed on to 'submit_le' (i.e., to submit to the background or to a queue system, respectively).

5.14 Restart from restart file

When a run has crashed at some point due to system problems, the restart file can be used to restart the simulation at the day where it ended, without spin-up. To use this option, change the date of the simulation in the main rc-file to the date of the last restart file. E.g. when the last restart file is LE_mysimulation_state_20120823_0000.nc, the day and time for the simulation should be 2012-08-23 00:00:00. Set le.restart : T, and make sure that the path and runid of the restart file are correct. In principle the run starts exactly on the daIt is safer but not mandatory to move the exitsing output directory to a different path (e.g. output2). Be aware that the .ctl files for GRADS are reinitiated so that they start now with the time stamp of the restart. This is easily modified in a text editor when one recombines the data from the original simililation and the restarted simulation.

6 External libraries

6.1 Required libraries

The following external libraries should be available: to compile the model:

- NetCDF library, including Fortran interface. Required for:
  - reading meteorological and other input data;
  - writing model output.

  The model does not use NetCDF-4 feautures yet; it is therefore sufficient to link with a NetCDF-4 library that was compiled without NetCDF-4 features enabled, or even with a classic NetCDF-3 library.
  Note that if the shared libraries are used, it is necessary to link a special Fortran interface:

  ```
  -lnetcdff -lnetcdf
  ```

  and eventually to tell the linker to add the path to the shared libraries to the runtime search path:

  ```
  -Wl,-rpath -Wl,${NETCDF_HOME}/lib
  ```

- Optional the HDF5 library is required, if the NetCDF library was compiled with NetCDF-4 features enabled. Needs the compression libraries too:
  - JPEG : probably available by default, link with -ljpeg;
  - Z : probably available by default, link with -lz;
  - SZ : probably installed together with HDF5, link with -lsz
- UDUNITS library, either version 1 or version 2. Required for unit conversions related to input files adopting CF-conventions. Version 1 has a Fortran interface, and allows the model to be compiled with a rather old compiler version too. For version 2 a binding to a C library is made, and this requires a compiler with F2003 support.

6.2 Configuration

To compile the model with the correct libraries, configurations are required at a number of levels.

- The user should explicitly enable the required libraries using macro definitions. For example, to enable NetCDF library on a system that has with libraries that have the NetCDF-4 features enabled:

  ```
  my.le.define : with_hdf4 with_hdf5
  ```

  See chapter 7 for details on macro definitions. The template settings should be ok for most system.

- The expert settings deterimine:
  - which libraries should be linked given the macro definitions; for example, if HDF5 is required also the compression libraries should be linked:

    ```
    build.configure.libs.ifdef.with_hdf5 : hdf5 sz jpeg z
    ```

  - the order in which the libraries should be linked.

See section 5.4.5 for details on the expert settings.

- The machine-specific settings specify the locations of the libraries; see section 5.4.3 for details.

7 Pre-processor macro's

Preprocessing macro's are a convenient tool in programming large applications. This chapter describes the use of these macro's in the LOTOS-EUROS source.

7.1 Expert settings

Working with macro's is considered an expert job, since small typo error could cause a part of the code to be omitted without noticing. Therefore, the expert rcfile includes lists of macro's that are supported. The source code is checked on use of macro's that are not supported; if these are found, an error is raised and configuration stops. The lists in the expert rcfile are used to add macro definitions to the proper macro include file.

7.2 Example of usage

Sometimes a minor modification of the actual source code is needed, for example because certain compilers cannot digest a piece of code, or to enable code that depends on an external library that might not be available (but is not always needed either). For this so-called pre-processing macro's are used. For example, the following code hides the use of the HDF4 library that is usually only needed for very specific input files:

```
#ifdef with_hdf4
  ! open hdf file:
  call HF90_Open( 'input.hdf', HF90_READ, hdf_id, status )
  ...
#else
  stop 'not compiled with HDF4 library enabled'
#endif
```

In this example, if the macro 'with_hdf4' is defined, then only the code between '#ifdef... ...' and '#else' is used, otherwise the code between '#else' and '#endif' is used. If the code is compiled with the HDF4 library enabled, then the macro 'with_hdf4' should be defined and the file '`input.hdf`' will be opened as specified, if this piece of code is called. However, a user wants to run the model, but does not need to read HDF4 files, the macro definition could be omitted. This is in particular useful if this library is not available; the user should not be bothered with a compiler complaining that a library is not available while it is not used anyway.

7.3 Macro definition include files

Macro's are defined by statements like:

```
#define with_hdf4
```

All macro definitions are collected in small include files. For example, for macro definitions in LOTOS-EUROS source files the include file 'le.inc' could look like:

```
!
! Include file with macro definitions.
!
#define with_hdf4
```

This file is included in the header of a file with:

```
#include "le.inc"
```

7.4 User settings

Which macro's are defined is something that is often user and application dependent. Therefore, a list of macro's to be defined is part of the rcfile:

```
my.le.define    :    with_hdf4
```

From the values in this list the macro include file(s) are written automatically by the source configuration scripts.

8 Input data

8.1 Minimum package

A package with input data is available to run the model with the default settings to verify proper installation. They can be downloaded from the SharePoint site. MACC III emission data is not included but can be obtained upon request. Meteorological data and boundary conditions from MACC cannot be provided by TNO.

8.1.1 Content

The content is as small as possible, but large enough for a model run with the following properties:

- simulation period August 2012;
- European domain as used for operational forecast;
- MACC-II emissions 2009);
- boundary conditions from climatology

The directory tree of the data is as follows:

```
models/
      /{\modeltype}/
                  /data/
                        /cf-standard/
                        /meteo/
                              /ecmwf/od/europe_w30e60n25n70_...
                        /landuse/
                                /forest/
                                /soiltext/
                                /traffic/
                                /soilwater_avg/
                        /ammonium/
                        /emissions/
                                  /\hl{MACC-II}/
                        /bound/
                        /standard/
      /GEMS/
            /rd/
            /mozart_out/
      /MACC/
            /fire/
observations/
            /AQORD/
```

8.1.2 Settings

The location of the input data is machine specific and therefore set in the 'machine.rc' file. With the data unpacked to a directory:

```
/data/models
```

use the following two settings to have the correct locations in the main rcfile:

```
MODELS          :   /data/models
my.data.dir     :   ${MODELS}/{\modeltype}/data
```

9 Horizontal grid

9.1 Grid definition

The horizontal grid is defined in the either the main rcfile or the regions rc file by specifying the lower-left corner, the resolution, and the grid size. The default grid is currently the same as used in the operational forecast:

```
grid.west  : -15.0
grid.south : 35.0
grid.dlon  : 0.50
grid.dlat  : 0.25
grid.nx    : 100
grid.ny    : 140
grid.nz    : 4
```

Input is interpolated or converted automatically to the required grid; if the model domain extends beyond the coverage of the input, an error is raised.

9.2 Zooming

Running in zoom modes simply means running the model twice: first for a large domain and coarse resolution, then for a smaller domain in fine resolution. The later should be configured to read concentrations from the first run as boundary conditions.

The following steps are usually taken to setup a zoom run.

1. Perform a run on large domain:
   - large domain covering the future zoom grid;
   - coarse resolution;
   - output of 3D concentration fields:
     - all relevant tracers;
     - bounding box around zoom region to save disk space (section 12.2.1);
   - remember the run-id and the output directory.
2. Perform the zoom run:
   - small domain within the previous domain;
   - fine resolution;
   - boundary condition type 'le', in main .rc file. Configure the settings in the main .rc file to read the (bounded) 3D concentration fields written in the previous run by referring to the right runid and location;
   - eventually put out concentrations in the halo cells to check the inheritance of the boundary conditions, see section 12.2.1.

Note that the time steps for the zoom run are automatically reduced to match the CFL (Courant)-criterion.

10 Meterological data

10.1 Introduction

The LOTOS-EUROS model uses off-line meteorology. Meteorological fields are read from files with time series of data at for example 3 hourly resolution.

The storage and reading of meteorological fields has been revised completely for OpenLE v1.0 and LOTOS-EUROS v1.10.007. The new implementation is based on general routines that are able to handle data files in NetCDF format following common conventions. At introduction of these versions, the only supported data files are retrieved from the ECMWF meteorology using scripts that accompany the model.

Previous versions of the model also supported meteorological data from the RACMO regional climate model and the WRF meteorological model. The new generic interface of the model will be extended to support data files produced by these models too.

10.2 Data definition in model

The meteorological data in LOTOS-EUROS is allocated dynamically, and only if actually needed. The dynamic allocation is part of generic facility in the model that is intended to hold all gridded variables, but as first step holds the meteorological variables.

Definition of the meteorological variables is done in the settings file(s), in particular:

```
lotos-euros-data.rc
```

The header of the files contains details of the possible settings, here we describe the main steps only. The first definition is a (long) list of all data variables supported by the model. This list contains for example a keyword to describe the 2m surface temperature:

```
data.vars     :  ... tsurf ...
```

For each of the variables, a detailed description should be provided in the form of a number of specific settings lines. For the surface temperature 'tsurf' this is for example:

```
! define:
data.tsurf.long_name          :  surface temperature
data.tsurf.units              :  K
data.tsurf.range              :  0.0 Inf
data.tsurf.gridtype           :  cell
data.tsurf.levtype            :  sfc
data.tsurf.datatype           :  instant_field_series
data.tsurf.input              :  meteo.tsurf
```

The detailed settings contain (see also the header of the settings file):

- a descriptive *longname*, used when the variable is put out;
- the units;
- the range of allowed values, used to truncate the variable to realistic values (sometime necessary to remove tiny negative values for example);
- the horizontal grid on which the variable is defined;
- the vertical levels on which the variable is defined;
- the datatype (Fortran class), which is mainly related to the temporal behavior (see section 10.6);

- a keyword that describes the input mechanism in case the variable should be read from file(s) (see section 10.7; otherwise, a definition of how to compute the field should be present (see section 10.8.

10.3 Accessing a variable in the model

In the model code, the variables that are defined in the settings are available from a module:

```
use LE_Data
```

The array that holds the variable values should be accessed using a pointer. Use the following code to access the above defined surface temperature:

```
use LE_Data only : LE_Data_GetPointer

! this will point to the surface temperature:
real, pointer                 :: tsurf(:,:,:)    ! (lon,lat,1)

! assign pointer to surface temperature,
! check if the units are as expected:
call LE_Data_GetPointer( 'tsurf', tsurf, status, check_units ='K')
IF_NOTOK_RETURN(status=1)

! show value in cell (3,4) :
print *, 'example of surface temperature: ', tsurf(3,4,1)
```

Note that:

- the variables are always 3D, for surface fields the last dimension has size 1;
- a check on the units can be used to make sure that the units are as expected (thus degrees Kelvin and not degrees Celcius);
- the call to the 'GetPointer' routine will return with an error status if the requested variable was not defined in the settings, or when it was not actually enabled (see section 10.4).

10.4 Enabling a variable in the model

Before a variable could be used, it should not only be defined in the settings, but also actually enabled. By enabling a variable it will be allocated and filled with the proper values at every time step.

Enabling is typically done in the initialization routine of a module, where all variables should be enabled that are used in the module. If a variable is not enabled, for example because some processes in the model are not used in a particular configuration, it will not be allocated (which saves memory) and will not be read or computed (saves run time).

To enable a variable, use the following lines of code:

```
use LE_Data, only : LE_Data_Enable

! enable surface temperature:
call LE_Data_Enable( 'tsurf', status )
IF_NOTOK_RETURN(status=1)
```

10.5 Grid types

The following grid types are supported:

- `cell`
  Values valid for the entire cell; this is the most common type. The shape of this grid is '(`nlon`,`nlat`)'.
- `corner`
  Values are defined on the corners of cells. This is used to setup the advective fluxes through the cell edges. The shape of this grid is '(`nlon+1`,`nlat+1`)'.
- `u-edge`, `v-edge`
  Used for fluxes through the west and east sides ('u'), or through the north and south sides ('v'). The shape of this grid is '(`nlon+1`,`nlat`)' or '(`nlon`,`nlat+1`)' respectively.

When fields are read from input files (section 10.7), a re-mapping to the target grid is performed automatically if possible. Current implementations can handle longitude-latitude grids (eventually with irregular spacing), and to some extend the so-called reduced grids which have a varying number of cells per latitude band.

10.6 Data types

The variable definition contains a setting for the datatype:

```
data.tsurf.datatype                : instant_field_series
```

The following data types are possible:

- `field`
  This type is used for a constant field, thus without any change in time, for example the grid cell area.
- `instant_field`
  An instant field is valid for specific instant time. This type is typically used for fields that are computed from other instant fields, thus not read from a time series in input files.
- `instant_field_series`
  Use this type for a variable that derived from a time series, typically by interpolation in time between the fields in the series. For example the 2m surface temperature is of this type.
- `constant_field`
  In this context a 'constant' field means 'constant during a time interval', for example during 3 hours. A time interval for which the variable is valid is associated with it. This type is typically used for fields that are computed from other constant fields, thus not read directly from file.
- `constant_field_series`
  Use this type to read 'constant' fields from a time series in a file. For example the amount of precipitation is of this type, since in meteorological data is available as the total amount of water that has fallen down during some time interval.

10.7 Input descriptions

If a variable should be read from input files, a configuration of the following form should be present:

```
data.tsurf.input                : meteo.tsurf
```

This setting describes that the details of how the field should be read are defined with settings starting with 'meteo.tsurf'.

For the current model version, meteo files were created from ECMWF data, see section 15.1 for creation scripts. Examples of input descriptions for these files are found in the settings file:

```
lotos-euros-meteo-ecmwf.rc
```

Here we show the general idea of the settings; for the specific settings we refer to the actual rc file.

The input description should first contain an 'input' value that defines time intervals and descriptions:

```
meteo.tsurf.input : 2012-01-01 00:00, 2012-12-31 23:59, descr1 | \
                    2013-01-01 00:00, 2020-12-31 23:59, descr2
```

That is, a list of time intervals and descriptions separated by a vertical line ('|') should be provided. Given a requested time, the input will be read according to the description associated with the interval that encloses the time value. This feature is used to handle changes in for example file formats and resolutions over time, since these often subject to evolution.

The input description takes the form of 'key=value' pairs separated by a semi-colon (';'). For example for the surface temperature it could have the form:

```
filename=/data/ECMWF/sfc/%{yyyy}/t2m_%{yyyymmdd}_3h.nc;\
long_name=2 metre temperature
```

The 'filename' value provides a template for the file name in which the variable should be found; the keywords '%{yyyy}' etc. are replaced by actual time values. The 'long_name' value identifies the file variable, in this case it should have the 'long_name' attribute with the provided value.

For instant fields (valid for a single time step), also the temporal interpolation method should be defined. Here we define that the expected temporal resolution in the files is 3-hourly, and that that requested fields should be interpolated linearly in time:

```
meteo.tsurf.tinterp         : interpolation=linear;step=3;units=hour
```

10.8 Computing variables

If a meteo variable should not be read from input files but computed from other variables, the definition should have a 'call' description. For example, the following description is used to define a variable 'srh' with relative humidity at the surface, which should be computed from the 2m surface temperature ('tsurf') and the 2m dewpoint temperature ('dsurf'):

```
data.srh.long_name              : surface relative humidity
data.srh.units                  : %
data.srh.range                  : 0.0 100.0
data.srh.gridtype               : cell
data.srh.levtype                : sfc
data.srh.datatype               : instant_field
data.srh.call                   : RelativeHumidityTD( tsurf, dsurf )
```

The 'call' description has the form of a function call, with as arguments a comma-seperated list of source variable names. The function name ('RelativeHumidityTD') is linked to an actual function call in subroutine:

```
Variables_Setup     (module LE_Data_Variables)
```

The arguments of the actual function are usually the name of the target variable ('srh') and a list with the names of the argument variables ('tsurf','dsurf').

If a variable with a 'call' definition is initialized, it is checked if all arguments are names of variables. When the variable is enabled, the arguments are enabled too.

10.9 Example: 3D field

The following example show the definition of a 3D field, in this case temperature. Such a variable combines most of features described in this chapter.

The input time series of temperature is here assumed to have a 3 hourly resolution, and defined on pressure levels. At every time step the temperature on model levels should be computed from a temporal interpolation between the time steps in the input, and a vertical remapping to the model levels (horizontal re-mapping is done at reading from input as described in section 10.5). For the vertical re-mapping it is necessary to have the level definition of the input fields and the model; here we apply an air-mass weighted re-mapping that requires level definitions in term of half-level pressures.

The name of the temperature variable in the model will be 't'. This will be computed from a variable 't_met' which holds the temperature field at the levels of the meteorological model (as read from the input). For the re-mapping also half-level pressures for the model and the meteorological input are needed. Therefore, in total 4 variables are needed in the model:

```
data.vars                : ... hp_met t_met hp t ...
```

The temperature field is defined with:

```
data.t.long_name         :  temperature
data.t.units             :  K
data.t.range             :  0.0 Inf
data.t.gridtype          :  cell
data.t.levtype           :  levels
data.t.datatype          :  instant_field
data.t.call              :  MassAverage( hp_met, t_met, hp )
```

The variable is defined on grid cells and model levels, and is an instant field valid for a single time step. At every time step, the values should be computed from a mass-weighted average from the temperature at the meteorological levels, which also requires half-level pressure fields.

The temperature at meteorological levels is defined with:

```
data.t_met.long_name     :  temperature
data.t_met.units         :  K
data.t_met.range         :  0.0 Inf
data.t_met.gridtype      :  cell
data.t_met.levtype       :  meteo_levels
data.t_met.datatype      :  instant_field_series
data.t_met.input         :  meteo.t
```

The data type is that of an instant series, since the field should be interpolated in time from a time series of input fields. The input is described by rcfile keys that start with 'meteo.t'. The level type 'meteo_levels' forces the model to maintain the original levels as read from the input files.

The input description of the temperature fields is for the ECMWF input used by the model looks like (see 'lotos-euros-meteo-ecwmwf.rc' for the exact form):

```
meteo.t.input    : 2012-01-01 00:00, 2020-01-01 00:00, \
                     filename=/data/ml-tropo20/t_%{yyyymmdd}_3h.nc;\
                     long_name=Temperature
meteo.t.tinterp  : interpolation=linear;step=3;units=hour
```

For the definition of the half-level pressure fields we refer to the settings in '`lotos-euros-data.rc`'.

11 Generation of chemistry code

11.1 Overview

With the number of tracers growing throughout the years it became more and more difficult to manage the source files solving the chemistry. In addition, there was no easy way to quickly enable or disable some tracers or chemical reactions without having to change the source code.

It was therefore decided to let some of the tracer and chemistry related files be generated by the scripts, based on settings in the rcfile. The scripts used to generate the source files start with the name 'genes' (GENErate Sources), and this name is also used in the settings keywords.

The generated files are:

- 'le_indices.inc', an include file with index parameters, names, units, and other settings for all tracers;
- 'le_chem_work.F90', a module with routines that fill the reaction rates and perform a single step of the iteration step in used by the chemistry solver.

Default versions of the generated source files are included in the base source. Editing won't have any effect; instead change the rcfile settings or the tables that were used to generate them, as explained below.

The new source files are generated in the build directory. For debugging it might be useful to take a look at the generated files after creation; this is facilitated by the rcfile setting:

```
genes.show.command : nedit ${genes.files} &
```

In this case, the new files are loaded into the 'nedit' editor; the ampersand ensures that the editor keeps running as background process while the rest of the model setup continues.

11.2 Tracer and chemistry properties

An important part of the configuration is done through so-called 'properties'. Table 11.1 shows some of the the currently supported properties; their use will be explained later on. A list with supported properties is maintained in the expert rcfile, which is used to check the settings made by the user; see the expert rcfile for a complete list of all properties.

To select the tracers and reactions that should be included in a simulation, a list with properties should be specified in the rcfile. Default selection in the main rc file is:

```
genes.prop.selected : cbm4 ppm ec pom sia seasalt dust accum biascorr
```

This is used to select the appropriate lines from the tables described below.

cbm4	tracers/reactions of CBM4 scheme
sulphur	sulphur-only scheme (SO2 and SO4a, OH read in)
methane	methane-only scheme (CH4, OH read in)
co2	CO2 tracer
sf6	SF6 tracer
nmvoc	non-methane volotile organic carbon
radical	radical
ppm	primary particulate matter
ec	elementary carbon
pom	primary organic matter
sia	secondary inorganic aerosols
seasalt	sodium aerosols representing seasalt
dust	dust aerosols
m7	M7 aerosol scheme
vbs vbs_cg vbs_soa	for secondary organic aerosol modelling
basecation	base-cat-ion aerosols
hm	heavy metals
accum	accumulated species
biascorr	bias corrected species
aerosol	aerosol tracer
fine_mode	fine mode aerosol
coarse_mode	coarse mode aerosol
all_modes	total aerosol collecting all size modes
in-cloud	used to select implicit in-cloud chemistry reactions

Table 11.1 Tracer and chemistry properties.

11.3 Tracer table

The core of the tracer selection is a table with all supported tracers. The default table is available as:

```
base/000/data/tracers.csv
```

The content looks like:

```
name , description                , units , formula , properties
NO   , Nitric oxide               , ppb   , N + O   , cbm4
O3   , Ozone                      , ppb   , O 3     , cbm4
ALD  , Aldehyde                   , ppb   , C 2 + R, cbm4 nmvoc
PPM_f, prim.part.mat. fine mode, ug/m3, R       , ppm aerosol fine_mode
:
```

For each supported tracer, the following values should be set:

- *name* Short name, used in index variables i_O3 etc.
- *description* Longer description, only used in comments in generated source files.
- *units* Units of the tracer in the model; usually 'ppb' for gasses, and 'ug/m3' for aerosols.
- *formula* Chemical formula (if possible). For some applications (labeling) it is useful to have at least the number of C, N, and S atoms in a molecule; for the remainder, use the symbol 'R' .
- *properties* List with all tracer properties (table 11.1) that are appied to this tracer.

To be enabled in the simulation, a tracer should have at least one of the selected properties.

11.4 Reaction table

Chemical reactions are also specified in a text file. The default table is available as:

```
base/000/data/reactions.csv
```

The content looks like:

```
label , reactants     , products        , rate expression              , properties
R1    , NO2           , NO + O3         , 1.0 x <NO2_SAPRC99>          , cbm4
R3    , O3+NO         , NO2             , 2.64 @ 1450                  , cbm4
:
RH1f  , SO4a_f + N2O5, SO4a_f + 2*HNO3, rk_het(ireac_N2O5_NH4HSO4a_f), cbm4 sia
:
```

For each reaction, the following values are specified:

- *label:* A short label for the reaction, used in parameter names.
- *reactions:* Tracers reacting with each other; tracer names following the tracer table.
- *products:* Tracer products.
- *rate expression:* Description of the reaction rate, see also Appendix A in the Reference Guide. See the comment in the top of the reaction table for how the expression is expanded. Some reaction rates are set by a special routine in the code, for example for the heterogeneous reaction 'RH1f' in the example lines above.
- *properties:* List with all tracer properties (table 11.1) that that should be present to have this reaction enabled.

Reactions are only enabled if ALL of it's properties are selected in the rcfile. Thus, reaction 'RH1f' of the example above is only enabled if both 'cbm4' and 'sia' tracers are selected.

11.5 Specification of table files

The name of the tables file should be specified in the rcfile. The default setting in the rcfile is:

```
genes.tracers.file      :   ../data/tracers.csv
genes.reactions.file    :   ../data/reactions.csv
```

In this case, the tables are found in the 'data' sub-directory next to the 'src' sub-directory in the build directory, thus:

```
<rundir>/build/data/
```

These table files are copied from the base directory, and eventually replaced by project specific versions. Therefore, to test a new table, simply put it in the 'data' sub-directory of a project directory. Alternatively, specify an absolute path in the expert rcfile settings.

11.6 Tracer indices and arrays

The information on the selected tracers is used to create the file 'le_indices.inc'. This file is included into 'le_indices.F90', which provides the module 'Indices'. Through this module, the user can access the generated entities; see the comment in top of 'le_incdices.F90' for their definition.

12 Model output

12.1 Output frequency

The user should specify the 'output' time-step in the rcfile:

```
! output time step in minutes:
timestep.output      :   60
```

A typical value is 1 hour. The model will arrive at every multiple of this output-time-step and put out simulated values.

12.2 Gridded output

The output of gridded fields is controlled by the rcfile. The supported output types are described below. It is possible to put out files of the same type but with different properties, for example files with concentration fields at the surface for many tracers, and files with 3D fields for some selected tracers only.

The gridded output is written to NetCDF files. By default the files are structured following the CF-conventions. GrADS description files are added for visualization (see section 13.1).

12.2.1 Concentration fields

For output of concentration fields the following properties could be set:

- which tracers to be put out:
  - model tracers;
  - accumulated tracers: total PM10, total carbon, etc; see indices.F90 for the supported accumulation;
  - bias corrected tracers;
- vertical axis:
  - model levels, including the concentration at 2.5m (which is the measurement height, denoted as surface concentrations, -sfc) and the top boundary layer;
  - heights relative to orography;
  - elevations relative to sea-level;
- whether to put out the cell height too;
- temporal resolution (typically fields are put out every hour);
- collection per file: either daily or instantaneous;
- horizontal coverage: by default the whole grid, but optionally:
  - bounding box to limit the horizontal area, for example to save storage space while producing boundary conditions for a zoom run;
  - halo cells (boundary conditions values);

12.2.2 Tracer total columns

Total columns could be put out for comparison with satellite observations. Note that this only includes the model layers now; the top boundary is not included yet, since usually there is no idea about its height. The following properties could be set:

- which tracers to be put out (normal or accumulated);
- temporal resolution (typically fields are put out every hour);
- collection per file: either daily or instantaneous.

12.2.3 AOD columns

The Atmospheric-Optical-Depth is computed from the tracer concentrations and put out as columns.

12.2.4 Model data fields

For output of various other model data fields (e.g. meteorological fields), the following properties could be set:

- which fields to be put out: meteo variables, stability fields, ...
- vertical axis:
  - model levels, including the surface level and the top boundary layer which are for most fields a copy of the nearby model layer;
  - heights relative to orography;
  - elevations relative to sea-level;
- temporal resolution (typically fields are put out every hour);
- collection per file: either daily or instantaneous.

12.2.5 Emission fields

The total emission for a certain tracer (or accumulated tracer) could be put at at regular times. The array that is put is 'emis_a' which contains for each tracer the total emission during the current (next) time step, independent of the source (anthropogenic, biogenic, sea-spray, etc). The following properties could be set:

- for which tracers the emissions should be put out, including accumulated tracers;
- vertical axis: either model layers or the total per grid cell;
- temporal resolution (typically fields are put out every hour);
- collection per file: either daily or instantaneous.

12.2.6 Dry and wet deposition

The dry- and/or wet-deposition budgets per output time step could be put out with the the following properties:

- for which tracers the deposition should be put out, including accumulated tracers;
- vertical axis: either model layers or the total per grid cell;
- temporal resolution (typically fields are put out every hour);
- collection per file: either daily or instantaneous.

12.2.7 Deposition parameters

Deposition variable such as resistances and deposition velocities could be put out with the following properties:

- for which tracers the variables should be put out (if tracer dependent);
- the landuse classes for which variables are valid;
- vertical height, for example needed for dry deposition velocities.

12.2.8 Daily budgets

The daily budgets are updated every time step and are put out at midnight. The following budgets could be put out:

- dry deposition flux of SOx, NOx, or NHx;
- wet deposition flux of SOx, NOx, or NHx;
- ozone dry deposition flux per landuse;
- ozone daily maximum;
- average NH3 concentration in soil.

12.3 Satellite validation output

For special purposes also the following fields could be produced:

- Simulation of OMI NO2 columns.
- Simulation of MODIS AOD columns.

The horizontal model grid is used to sample the satellite pixels. This output therefore requires that information on the satellite pixels is available.

12.4 Observation simulation

The standard method to produce simulations of concentrations at observation sites is to extract them from the (2D or 3D) gridded output.

An alternative is to use the MAORI (Model And Output Routine Interface) routines. The MAORI interface is configured via the rcfile by specification of a table file with station location and settings to define the simulated tracers etc. This is mainly used in the Kalman Filter context to simulate state values at observation sites; in the default model the code is present, but the use is not fully supported yet.

12.5 Emission summary

A summary of the emissions is written automatically to the output directory. Currently this is only supported for anthropogenic emissions read from files in TNO format. Since these are year-dependent, the summary is written for every year that the simulation covers. A summary consists of four comma-separated-value file (plain text): a list of the emission categories, a list with country codes and names, a table with total emissions for a component per country, and similar per country and emission category. The file is useful for verification of the emission input.

13 Post processing and visualization

Post-processing and visualization of the model output is often very project and user specific. The netCDF file format compiliant with conventions ensures that many packages can be used (e.g. Matlab, Python, IDL). In this chapter we give an overview of the available standard tools, which will help a user with the first steps.

13.1 GrADS

The *Grid Analysis and Display System* (GrADS http://www.iges.org/grads) is an interactive desktop tool that is used for easy access, manipulation, and visualization of earth science data.It is very suitable to quickly browse through the data without scripting and perform a fist screening of the output.

13.1.1 Control files

LOTOS-EUROS supports the use of GrADS by adding control files (`.ctl`) to the output directories that are used by GrADS to read the output and define the correct grid, time range, etc.

13.1.2 Running

If GrADS is installed, use the following command to start:

```
grads
```

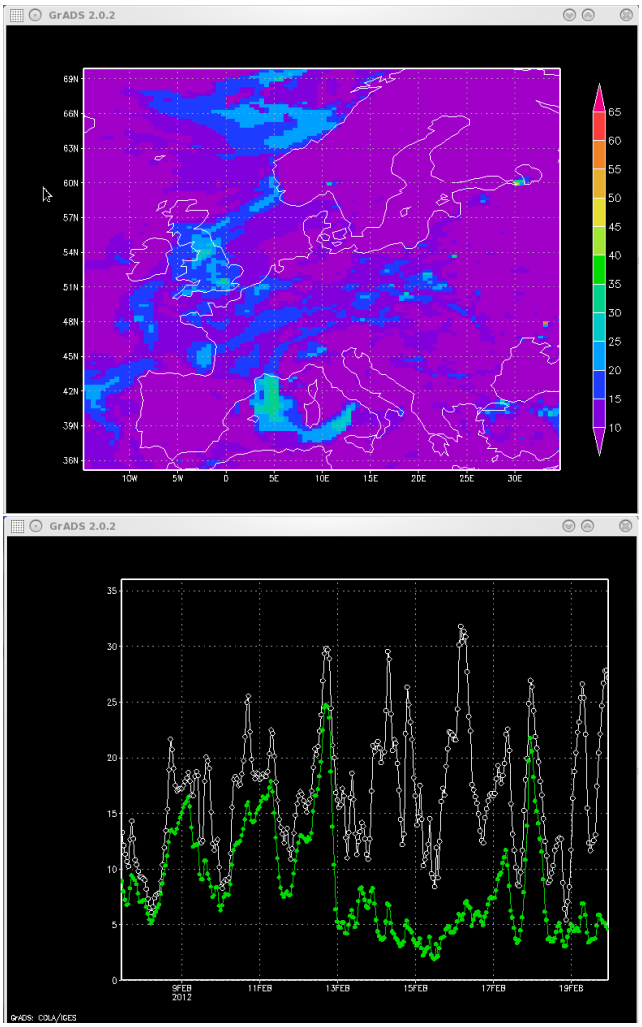

Figure 13-1 Screenshot of a GrADS graphics window with a snapshot of modelled PM10 concentrations and timeseries of PM10 and PM2.5 in gridcell corresponding to Utrecht (Netherlands)

14 Run time

14.1 Timing

The time spent on different tasks in the model is continuously measured.

A timing profile is written to a text file with extension '.prf' in the output directory. In the header of this file, a pretty print of the absolute and relative time spent on a task and its sub-tasks is shown:

```
timer                                    system_clock        (%)
...
model time loop                             11504.66
  time step output                            712.27 (   6.2 %)
  time step setup                            1223.58 (  10.6 %)
  adjust                                       49.89 (   0.4 %)
  advection                                   573.52 (   5.0 %)
  chemistry                                 68110.19 (  59.3 %)
  vertical diffusion                         1144.94 (  10.0 %)
  dry deposition                              217.74 (   1.9 %)
  sedimentation                                24.93 (   0.2 %)
  wet deposition                               23.61 (   0.2 %)
  emission                                     35.59 (   0.3 %)
  other                                       679.41 (   5.9 %)
....
```

14.2 Parallelization

14.2.1 OpenMP

The LOTOS-EUROS model uses OpenMP to run in parallel. To run on multiple threads, enable the 'par.openmp' flag in the rcfile and specify the number of available threads. It is our experience that usually 3 to 4 is a good choice, more than 8 will not lead to a further decrease of total runtime.

15 Tools

The LOTOS-EUROS 'tools' are additional software packages next to the model.

For OpenLE, the tools are found in the '`tools`' directory next to the base source (see section 4.2).

15.1 Create files with ECMWF meteorological data

Scripts to extract OpenLE meteorological data from the ECMWF archive, and to transfer the files to your local computer are present in:

```
tools/meteo/
```

An ECMWF member state account is required to run the scripts.

15.1.1 Extract data from ECMWF archive

The best (fastest) way to extract meteorological data from the ECMWF archive is to retrieve it from the MARS archive on the member state server (`ecgate`). After copying the '`tools/meteo`' directory to `ecgate`, edit and run the following script to create data files:

```
./bin/OpenLE_meteo_ecmwf
```

This will first retrieve files from MARS in `grib` format, and then convert to `netcdf`. The standard `grib_to_netcdf` tool at `ecgate` does not provide completely CF-compliant files, and therefore small additional modifications are needed.

Variables that are originally accumulated from the start of a forecast are converted to averages over an interval. For example, precipitation fields are originally stored in 'm water equivalent' since start, but converted to 'm/s' average during a (3 hourly) time interval.

The data volume is reduced significantly by combining vertical layers. How to combine levels is defined in the header of the script; for OpenLE we created files with about 20 layers in the troposphere. To combine the levels a Fortran program is compiled that is present in the 'src' directory. Compilation settings are defined in:

```
rc/openle_meteo.rc
```

15.1.2 Transfer meteo files from ecgate to local computer

Meteo files produced at `ecgate` could be transferred to a local computer using the script:

```
./bin/OpenLE_meteo_get
```

The script uses the '`ECaccess`' tools to scan a source directory on `ecgate` and get the files to the local machine if not present yet. Edit the settings in the top of this script to change locations etc.

16 Coding conventions

How should new parts of code be written ? Some ideas.

- Files contain either 1 module or 1 main program.
- Model specific files have a prefix equal to the model name:

```
le_data.F90
le_process.F90
    :
```

- Modules have the same name as the source file:

```
module LE_Process
  ...
end module LE_Process
```

- Module names (thus file names) should reflect the content:

```
! Data that are required for many different modules
module LE_Data

  ...
end module LE_Data
```

- The tasks to be performed by a module could be distributed over a number of sub modules:

```
              LE_Data                 # top module

 LE_Data_nc   LE_Data_hdf   ...      # specific

            LE_Data_Base              # shared entities
```

  In this case, other model routines should access the entities only via the top module:

```
use LE_Data, only : LE_Data_Setup
use LE_Data, only : temper, humid, tsurf
```

- Public routines in a module should start with the module name:

```
module LE_Data

  private
  public  ::  LE_Data_Setup
  ...

contains

  subroutine LE_Data_Setup( t1, t2, status )
    ...
  end subroutine LE_Data_Setup

  ...

end module LE_Data
```

- If a module defines public data, a module initialization and finalization routine should be provided. These routines might be dummy to be prepared for future extensions.

```
module LE_Data

  private
  public  ::  the_answer
  public  ::  LE_Data_Init, LE_Data_Done
  ...

  integer     ::  the_answer

contains

  subroutine LE_Data_Init( status )
    ...
    ! something to be done:
    the_answer = 42
    ...
  end subroutine LE_Data_Init

  subroutine LE_Data_Done( status )
    ! nothing to be done.
  end subroutine LE_Data_Done

  ...

end module LE_Data
```

- If a module defines a public type rather than public data, its name should start with the prefix 'T_' followed by the module name. An initialization and finalization routine should be created, eventually doing nothing:

```
module RcFile
  ...
  private
  public  ::  T_RcFile, RcFiLE_Init, RcFiLE_Done
  ...
  type T_RcFile
    integer  ::  id
    ...
  end type T_RcFile

contains

  subroutine RcFiLE_Init( rcf, fname, status )
    type(T_RcFile), intent(out)      ::  rcf
    character(len=*), intent(in)     ::  fname
    integer, intent(out)             ::  status
    ...
    ! something to be done:
    rcf%id = 123
    open( unit=rcf%id, file=trim(fname), form='formatted', iostat...
        ...=status )
    ...
  end subroutine RcFiLE_Init

  subroutine RcFiLE_Done( rcf, status )
    type(T_RcFile), intent(inout)    ::  rcf
    integer, intent(out)             ::  status
    ...
    ! something to be done:
    close( rcf%id, iostat=status )
    ...
  end subroutine RcFiLE_Done

  ...
```

```
end type RcFile
```

- Subroutines return an integer status value:

```
! return status:
!   <0  : warning
!    0  : ok
!   >0  : error

subroutine Process_Init( status )
  integer, intent(out)    ::  status
  ...
  ! ok
  status = 0
end subroutine Process_Init
```

- Functions are '*pure*' and preferably '*elemental*'.

Bibliography

Manders, Astrid, Arjo Segers, and Sander Jonkers (2016). *LOTOS-EUROS v2.0 Reference Guide*. TNO Report R10898. TNO. URL: www.lotos-euros.nl.

17 Signature

Name and address of the principal

Names of the co-operators
A.J. Segers
A.M.M. Manders-Groot
S. Jonkers

Period in which the research took place
January-June 2016

Name and signature reviewer

Prof. Dr. M. Schaap

Signature

Dr A.M.M. Manders-Groot
Project leader

Signature

Ir. R.A.W. Albers MPA
Research Manager