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

LOTOS-EUROS User Guide

v2.3.000

LOTOS-EUROS User Guide v2.3.000

Date	December 2023
Author(s)	Arjo Segers Astrid Manders Richard Kranenburg
No. of pages	65 (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. (2023).

The User Guide contains the following chapters.

- Chapter 2 describes the various websites related to the model.
- How to install the model is described in Chapter 3 on the version control system, and Chapter 4 on the structure of the source files.
- How to compile and run the model is described in Chapter 5. This requires proper configuration of paths to data and run directories, compiler names, library locations, etc.; details are discussed in Chapter 6 on libraries, Chapter 7 on pre-processing macro's, and Chapter 8 on input data.
- Chapters 9 on the grid, 11 on meteorological data, 13 on the chemistry code, and 15 on output are useful when a user would like to adapt the configuration of the model.
- The final chapters discus more in-depth elements such as employing parallel computation (17), post-processing tools (18), additional tools (19), and coding conventions (21).

1.2 Raw documentation

The original text of this User Guide is written in LaTeX format.

A pdf version can be created with:

```
make pdf
```

or, for the open-source release:

```
make open−pdf
```

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:

`airqualitymodeling.tno.nl/lotos-euros`

The website contains general information, links to the operational forecasts, documents, an overview of publications, and information on how to obtain the open-source version.

2.2 Local documents

Internal documents including validation reports can be found on the MODAS SV:

```
/tsn.tno.nl/Data/SV/sv-059025_unix/models/LOTOS-EUROS/doc
```

3 Version control system

3.1 Introduction

The source files of the LOTOS-EUROS model are managed with a version control system. With such a system users can obtain an up-to-date copy of the files, archive their changes, obtain changes made by other users. Currently 'git' is used for this.

Throughout this User Guide some examples will be given on how to use the git package to manage files. By default we assume that the 'git' program is a available, which is a standard tool in a Linux environment.

3.2 GitLab server

A GitLab site provides a web interface to browse through the files hosted under git. In addition, the site offers a wiki, issue tracking, and other features for devellopment.

The TNO internal devellopment version of LOTOS-EUROS is hosted at this gitlab site:

https://ci.tno.nl/gitlab/modas/lotos-euros

3.3 Checkout a copy of the source

To obtain a fresh and up-to-date source code, checkout a complete version from the server:

```
git clone git@ci.tno.nl:modas/lotos-euros.git
```

3.4 Managing source files with Git

To learn more about using Git, got to the wiki page on the GitLab site for first steps.

Some quick steps:

- To see if you changed some files in or below the current directory:

  ```
  git status
  ```

- To import changes made by others:

  ```
  git fetch
  git pull
  ```

- To add a new file to the repository:

  ```
  git add newcode.F90
  ```

- To commit a change to the server:

  ```
  git commit -m 'This is an important change.' newcode.F90
  ```

- To push your commits it to the repository such that other users can pull it:

```
git push
```

4 Source files

4.1 Identification of a model version

Each model version is given a three-leveled key, for example ”v2.3.000”. Here, the first 2 numbers identify the base version number (”v2.3”) 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. Checklists for how to create new versions and patches is left for chapter 20.

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/v2.3/                  # Version directory
                 README            # first aid information.
                 base/             # base sources
                      000/         # patch directories
                      001/         #
                        :          #
                 proj/             # modifications to the base source

tools/                             # Other (general) tools
      ttb/
      diadem/
      :
```

The ’tools’ directory is described in chapter 19, for the model only the ’lotos-euros/v2.3 directory is important.

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
base/003/    # ...
       :
```

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 5.5). For this example it will look like:

```
my.source.dirs        :  base/000 \
                         proj/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/v2.3
   ```

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

   ```
   base/000/rc/lotos-euros.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, compile an executable, and run or submit the job script(s) using:

   ```
   ./base/000/bin/setup-le  base/000/rc/lotos-euros.rc
   ```

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/v2.3
```

5.3 Run scripts

The script(s) used to setup, start, and submit a LOTOS-EUROS run are placed in:

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

The 'setup-le' script is the main script for each run.

5.4 Python files

Tools used to setup a simulation are placed in:

```
base/000/py/
```

5.5 Rc-files

The configuration of a model run is done through a text file called the 'rc'-file. The abbreviation 'rc' comes from '*resource*', or actually the Unix '*X-resource* file on which the format is based. However, also '*run configuration*' is a suitable expansion.

5.5.1 *Overview of rc-files*

The top-level rc file '`lotos-euros.rc`', contains the definition the run id, selects a version the base source and eventually project code, defines a time range and domain, selects emissions and boundary conditions, etc. To keep the main file as short as possible, detailed configurations are included from other rc files, for example:

- `lotos-euros-regions.rc` in which several model domains are predefined;
- `lotos-euros-landuse.rc` in which settings and file names for the land use files are defined;
- `lotos-euros-meteo.rc` in which paths and settings for input meteorology are defined;
- `lotos-euros-emissions.rc` in which paths and settings for anthropogenic emissions are defined;
- `lotos-euros-bound-*.rc` files in which paths and settings for several sets of boundary conditions are given;
- `lotos-euros-output.rc` in which one can specify the species and fields that have to be put out.

Also a number of build/compile/run settings are need; these have been distributed into:

- `lotos-euros-jobtree.rc` that defines the jobs that together perform a simulation, e.g. to build, initialize, and run a job.
- `lotos-euros-build.rc` in which macros and supported species are defined (section 5.5.6).
- `machine-*.rc` files with system-specific settings like compiler and libraries;
- `compiler-*.rc` files with compiler specific flags;

The top-level rc file `lotos-euros__anywhere.rc` contains with default settings for running outside the European domain. It configures a run to use land use and emissions sets that are available globally, but might be less detailed as what has been collected for Europe.

5.5.2 *Start: copy from template*

To configure your own run, best is create a sub-directory to store your settings:

```
mkdir -p proj/mytest/000/rc
```

This defines that for your project '*mytest*' you will use patch '‘\modelpatch‘'. The sub-directory 'rc' is used to keeps settings seperated from source files (that you might want to change too in future).

Then make a copy of one of the template top-level rc-files:

```
cp  base/000/rc/lotos-euros.rc   proj/mytest/000/rc/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, use of restart files, etc. 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.

5.5.3 *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 see the header of:

```
base/000/py/rc.py
```

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.

5.5.4 *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. A template is available:

```
base/000/rc/machine-template.rc
```

For each institute/machine combination, a machine-specific rc-file should be created. The settings are included in the top-level rc-file via:

```
! include settings:
#include base/${my.patch.nr}/rc/${my.machine.rc}
```

When GNU environment modules are used on your computer, the machine settings might be used to insert ’module’ commands in the top of job scrips. For the TNO server this is for example used to load the correct environment:

```
! modules loaded in job scripts:
*.modules            :   purge ; \
                         load slurm/18.08.8 ; \
                         load gcc-suite/8.2.0 ; \
                         load openmpi/4.0.5 ; \
                         load netcdf-c/4.7.4 ; \
                         load netcdf-fortran/4.5.3 ; \
                         load udunits/2.2.26
```

See chapter 6 for information about the external libraries that should be configured in the machine rc file.

5.5.5 *The compiler rc-file*

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

```
! compiler specific settings:
#include base/${my.patch.nr}/rc/compiler-gcc-v5.3.0.rc
```

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

5.5.6 *The build 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 'build' rc-file, which is included into the main rc file:

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

Do not modify this file unless you know what you are doing!

5.6 Setup script

To setup a run, first go to the model version directory:

```
cd lotos-euros/v2.3
```

Call the *setup* script with the rc file as argument to create a run-directory and compile an executable:

```
./base/000/bin/setup-le    proj/mytest/000/rc/lotos-euros.rc
```

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

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

5.7 The job-tree

The 'setup-le' script will start a squence of jobs to initialize and run a LOTOS-EUROS simulation.

By default the jobs are:

- le.copy job will copy the source codes to a run directory;
- le.build job will compile an executable;

- le.run job will run the executable, this is the actual simulation;
- le.post job will post-process the output (if necessary).

The list of jobs to be created is defined in:

```
lotos-euros-jobtree.rc
```

with:

```
jobtree.le.elements              :  copy build run post
```

The 'le.copy' job will always run in the foreground.

The 'le.build' job should probably run in foreground too, since this helps to quickly see compilation errors. This is defined in:

```
lotos-euros.rc
```

with:

```
! copy job is running in foreground (see "lotos-euros-jobtree.rc"),
! enable the following line to have also the build job in foreground:
jobtree.le.build.script.class    :  utopya.UtopyaJobScriptForeground
```

The default destination for the 'run' and other jobs is probably to submit it to a queue system. Since the queue system is diffent on each platform, the destination is defined in the machine-specific settings (section 5.5.4). For the TNO platform the SLURM queue should be used:

```
! default class with the job script creator:
*.script.class               :  utopya.UtopyaJobScriptBatchSlurm
```

This defines that the header of a job file should have comments with SLURM job options. See the examples of machine specific settings for inspiration on configure job settings for your own queue system.

5.8 The 'le.copy' job

This job is the first that is performed and will create a run directory with a copy of the source code.

5.8.1 Run directory

The first step taken by the 'le.copy' job 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}
```

5.8.2 Run-time rc-file

The 'copy' job will also create a *run-time* rc-file. This a single settings file with all includes and variable substitutions resolved; see section 5.5.3 for details.

The run-time rc-file is put in the directory:

```
<rundir>/run/lotos-euros.rc
```

5.8.3 *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 5.9.3 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 The 'le.build' job

The 'le.build' job configures the source files and compiles an executable.

5.9.1 *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 13);
- 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.

5.9.2 Creation of the Makefiles

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

5.9.3 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-debug check-all debug
```

The actual flags assigned to these keywords are set in the compiler rc-file described in section 5.5.5.

5.10 The ’le.run’ job

This is the actual run.

The executable is started either directly, or via an MPI started to have the correct environment for the domain decomposition.

5.11 The ’le.post’ job

This job is currently only used to compress output files. The compression is done using the ’NCO’ tools.

6 External libraries

6.1 Library settings

The location of external libraries such as NetCDF is strongly dependend on the machine where the model is compiled. Configuration is therefore done in the machine settings file described in section 5.5.4.

6.2 Librararies used in base code

The following external libraries are currently used, of which NetCDF is the only obligatory.

6.2.1 NetCDF library

This library is 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.

The machine-specific settings (section 5.5.4) specify the compile and link flags for this library, for example:

```
compiler.lib.netcdf.fflags    : -I/opt/include
compiler.lib.netcdf.libs      : -L/opt/lib -lnetcdff -lnetcdf
```

Note that in case shared libraries are used (as in this example), then it might turn out to be necessary to tell the linker to add the path to the shared libraries to the runtime search path:

```
compiler.lib.netcdf.libs      : -L/opt/lib -lnetcdff -lnetcdf \
                                -Wl,-rpath -Wl,/opt/lib
```

6.2.2 UDUnits library (optional)

The UDUNITS library is used to check if units read from a file match with the units used in the model. Some common unit comparisons have been hard-coded, so that for example for the test input provided with the open-source version the UDUNITS library is actually not necessary. In that case, the need for a UDUNITS library could be disabled in the machine rcfile (section 5.5.4) using the empty setting:

```
my.udunits.define       :
```

If the model complains however that a certain unit check cannot be performed, either hard-code a comparison on the location identified by the error message, or enable the need for a UDUNITS library. The UDUnits library might be available as version 1 or 2; since the interface is rather different between these, it was necessary to distuinguish between them in the code. Therefore, the flag in the machine settings should define explictly which version is available on the system:

```
 !~ version 1 is available:
 !my.udunits.define      :   with_udunits1
 !~ version 2 is available:
 my.udunits.define      :   with_udunits2
```

Note that for version 2 a C-binding in used that requires a compiler with F2003 support.

6.2.3 *LAPack library (optional)*

The LAPACK library is needed by the labeling code is enabled. When using Intel compiler suite, the optimized Intel Math Kernel Library (MKL) should be used.

6.3 Adding new libraries

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

- The user should explicitly enable the required libraries using macro definitions. For example, to compile with the HDF5 library, introduce a new macro 'with_hdf5' following the instructions in chapter 7.
- Ensure that this macro is defined by adding it to the 'define' list in the expert settings (section 5.5.6):

  ```
  build.configure.macro.define    : with_netcdf with_hdf5
  ```

- For this macro, provide a list with library names that should be linked with. For example, if HDF5 is required, then also a number of compression libraries should be linked. For this, the expert settings should contain a definition similar to:

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

- The order in which the libraries should be linked is defined by:

  ```
  build.configure.libs.all        :  netcdf hdf5 sz jpeg z
  ```

- The machine-specific settings (section 5.5.4) specify the compile and link flags for a library, for example:

  ```
  compiler.lib.hdf5.fflags    : -I/opt/include
  compiler.lib.hdf5.libs      : -L/opt/lib -lhdf5_hl \
                                   -lhdf5_fortran -lhdf5
  ```

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 Test 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:

```
inputdata/
          /ammonium/
          /bound/
          /cf-standard/
          /emissions/
                     /CAMS/REG/v2_2_1/
          /LEIP/
              /europe_w30e70s5n75/
                                   /ECMWF/od/   # meteo
                                   /MACC/fire/
          /landuse/
                   /forest/
                   /soiltext/
                   /soilwater_avg/
                   /traffic/
          /standard/
```

8.1.2 Settings

The location of the input data is machine specific and therefore set in the 'machine.rc' file:

```
my.data.dir      :   /data/inputdata
```

8.2 See also

For more detailed information on certain types of input data, see:

- chapter 11 on meteorological data;
- chapter 12 on boundary conditions.

9 Horizontal grid

9.1 Grid definition

Two different grid types are supported: a regular longitude/latitude grid (”*cartesian*”), and a grid that is curved in longitude/latitude sense (”*non-cartesian*”).

9.1.1 Cartesian grid

The default grid for most runs is regular in longitudes and latitudes, thus with equal spacing in degrees in both directions. In longitudes/latitudes the domain is square (figure 9-1, left panel).

The rcfile should first specify the grid type:

```
! define grid type:
grid.type           :  cartesian
```

For this grid type, the settings should then specify the lower-left corner, the resolution, and the grid size:

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

Template grid definitions are available in:

```
lotos-euros-regions.rc
```

If this file is included in the main rcfile, the grid definition to be used should be selected using a keyword. For example, the following settings will enable the ’MACC-II’ grid definition as used in the operational CAMS forecasts:

```
! select grid used by operational forecasts:
grid.name             : MACC-II

! grid definitions:
#include base/${my.patch.nr}/rc/lotos-euros-regions.rc
```

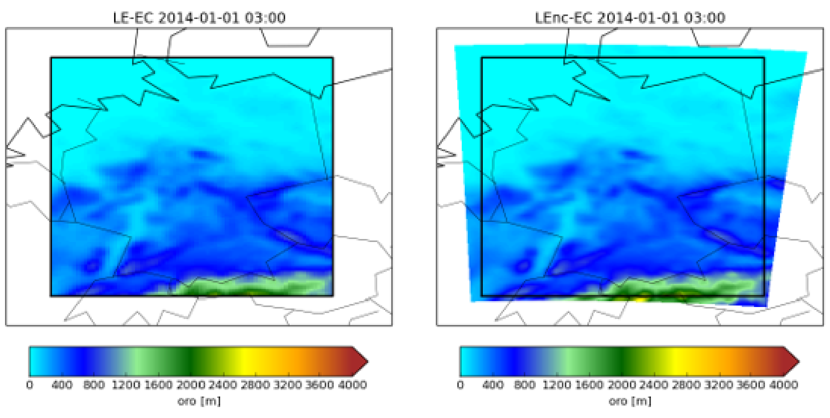


Figure 9-1 Examples of model extend on cartesian grid (left) and non-cartesian grid (right).

9.1.2 Non-cartesian grid

A non-cartesian grid is typically used when the model is driven by meteorological data from WRF or COSMO. In that case it makes sense to use the same grid definition as the meteorological model. In longitudes/latitudes the domain is usually wider towards the poles to have grid cells that have equal width in km (figure 9-1, right panel).

In the rcfile, first specify the grid type:

```
! define grid type:
grid.type          :   non-cartesian
```

The grid definition should then be read from a sample file with the WRF or COSMO meteo. The settings should specify the sample file name and the name of a variable from which the grid definition will be read, for example:

```
! sample file:
grid.file.name           :   /data/WRF/wrfout_d02_2014-01-01_00:00:00
! sample variable:
grid.file.var            :   var_name=HGT
```

In practice it is often only necessary to run the model on a sub domain of the input data. The following setting could be used to select this sub domain, or to specify that the full grid should be read:

```
! subset [i1,i2,j1,j2], negatives for all :
!grid.file.subset          :   -999 -999 -999 -999
grid.file.subset           :   50 110 50 150
```

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; use keyword: ‘all-advected’
        - bounding box around zoom region to save disk space (section 15.2.1);
        - conc-bound output; conc-bound output generate concentrations + pressure and height information of grid cells
    - remember the run-id and the output directory.
2. Perform the zoom run:
    - small domain within the previous domain;
    - fine resolution;
    - boundary condition; in main .rc file: Load bound-rc file including settings to use LE-run as boundaries. (lotos-euros-bound-*-le.rc). ‘conc-bound‘ output will be interpolated to zoom-domain
    - parent domain; Domain name of the boundary run
    - eventually put out concentrations in the halo cells to check the inheritance of the boundary conditions, see section 15.2.1.

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

In the setting of the rc-file the type of run can be given (‘my.run.type‘), the necessary outputs will be defined by default.

- Bound: run will be used as boundary run; keyword ‘all-advected‘ is used to get all advected tracers in output (conc-bound files).
- Zoom: run will be used as zoom run; Parent domain should be defined, rc-file including LE-settings will be loaded.
- Zoom or Bound_plus_Zoom; Run is used as both boundary and zoom run (middle domain in a three-step nested approach).

9.3 Mapping of input data

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

10 Vertical levels

Three different vertical level definitions are supported. Table 10.2 shows selected properties for the different definitions discussed below.

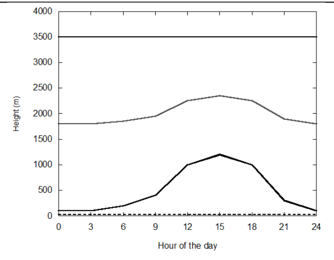
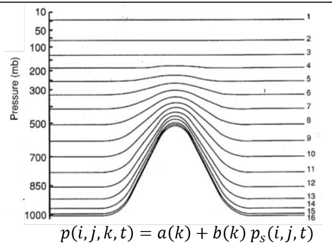
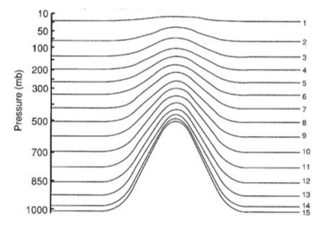
mixlayer	hyblevel	metlevel
		

Table 10.1 Illustration of level definitions.

levels.type	mixlayer	hyblevel, metlevel
number of layers	4 or more	unrestricted
vdiff.kz_type	`msp` (default) `normal` (optional)	`normal`
adjust process	yes	no
volumes	constant within time step (1 hour)	dynamically

Table 10.2 Properties of vertical level definitions.

10.1 Mixed-layer definition

The '*mixed-layer*' definition is the standard method for the vertical structure in LOTOS-EUROS. The levels are defined every time step in terms of heights above the surface, and some of them evolve dynamically with the boundary layer height obtained from the meteorology (figure 10.1, left panel). Input data will be mapped to this layer definition.

In a standard configuration with 5 layers, the first layer is thin surface layer of 25 m. On this the second layer is placed with the top at the boundary layer; this is therefore grid cell and time depended. The third and forth layer are reservoir layers of equal thickness of at least 500 m, with the forth layer having a top at 3500 m above the surface or more if needed to have the minimal thickness. On top of this the fifth layer is placed with a top at 5000 m or more if needed to have a minimum thickness of 500 m. The minimum thickness is increased with a factor times the orography of surrounding cells to have somewhat thicker cells over mountain area's; by using thicker cells, the minimum time step implied by the advection is higher which saves run time.

This level type is enabled using:

```
levels.type            :   mixlayer
```

The number of layers to be used is defined with:

```
levels.nz              :   5
```

The thickness of the surface layer is defined with:

```
mixlayer.surf_top      :   25.0
```

The mixing layer should be at least as thick as the surface layer, which is defined by a minimum value for the top above the surface:

```
mixlayer.mix_topmin    :   50.0
```

The fifth and higher layers are defined using lists that define the top above the surface, the minimum thickness, and the factor to be used in combination with the standard deviation of the orography. For the 5 layer version the configuration is:

```
mixlayer.top           :   3500.0  5000.0
mixlayer.dmin          :   500.0
mixlayer.sdofac        :   1.0  0.5
```

For an 8 layer version the configuration looks like:

```
mixlayer.top              :   3500.0  5000.0  6500.0  8000.0  10000.0
mixlayer.dmin             :   500.0
mixlayer.sdofac           :   1.0  0.5  0.0  0.0  0.0
```

A final configuration is the definition of the thickness of the aloft layer that will hold the top-boundary concentrations:

```
mixlayer.daloft                  :   1000.0
```

With this vertical scheme, one of the model operators is an 'adjust' step that changes the layer heights towards a new boundary layer height. This is applied at the start of every time step. During the time step, the grid cell heights (and thus cell volumes) are kept constant.

For diffusion between two layers two options are available to compute diffusion coefficients K_z at the layer interfaces. The first is the 'msp' method that has been used traditionally, which adjusts the K_z values for differences in layer thickness around the interface; this is enabled with:

```
vdiff.kz_type  :  msp
```

Alternatively also the 'normal' method could be enabled that is used by the other level definitions.

10.2 Hybride layer definition

A hybride-sigma-pressure scheme defines vertical layers using pressure boundaries that follow the surface pressure at the lower levels, and have fixed pressure boundaries at the top. The layers therefore follow the orography near the surface, which evolves slowly to layers at fixed pressure higher up (figure 10.1, middle panel). This scheme is for example used by the ECMWF meteorological model, and is therefore useful if LOTOS-EUROS is driven by this data. Input data will be mapped to the chosen layer definition anyway, regardless whether this is also a (different) hybride definition, or something else.

At every time step, the the pressures at a layer interface ar

$$p(i,j,k,t) \;=\; a(k) \;+\; b(k)\,p_s(i,j,t) \tag{10.1}$$

In here, (i,j) denotes the horizontal cell indices, k the layer interface index, and t the time. The interface coefficients a and b define for each layer which part fraction of the pressure is constant and which fraction depends on the surface pressure p_s that is part of the meteorological input.

The hybride level definition is enabled using:

```
levels.type            :   hyblevel
```

The number of layers to be used is then defined with:

```
levels.nz              :   12
```

The rest of the atmoshpere (above the layers simulated by the model) is filled with concentrations from the global boundary conditions; these are used for top boundary conditions, and to simualte satellite observations. Specify the total number of layers with:

```
levels.nz_top          :   18
```

NOTE: *the first top layer above the model layers should not be too thick! This will be filled with concentrations from the global boundary conditions, averaged over the layer. If the layer is too thick, the ozon concentrations will include part of the stratospheric ozone layer, and the inflow of ozone from the top will be too high!*

The hybride coefficients a and b should be provided in a text file, for which an example is provided with the model distribution:

```
hyblevel.coefficients    :   data/hyblevel-137_CL42__ml12p06
```

The number of coefficients in the file might exceed the requested number; the model will simply read the first values that define the interfaces from surface to model top.

In this scheme, the layers are first defined in terms of pressure boundaries, which then implies the heights above the surface (using temperature, humidity, and orography). The cell volumes change dynamically but do not follow the evolution of the boundary layer, and therefore no '`adjust`' process is needed as used for the '`mixlayer`' scheme.

Diffusion coefficients between layer interfaces should be normal K_z values, thus not adjusted for extreme differences in layer thickness:

```
vdiff.kz_type        :  normal
```

10.3 Meteo level definition

With this method, the model will copy a level definition from a set of meteorological data. Depending on the data this definition could define layer interfaces as pressures or heights above the surface. The model data settings as described in chapter 11 should define the correct settings to read and process the data. This method is useful to keep LOTOS-EUROS as close to a meteorological model as possible. The '*meteo-level*' definition is therefore a generalization of the '*hybride-level*' method, which does not necessarily require a surface pressure and hybride coefficients but could use other level definitions too (figure 10.1, right panel).

The meteo level definition is enabled using:

```
levels.type            :   metlevel
```

The layers to be used are defined with with two settings: the number of layers, and for each of these, the number of original layers that is coarsed in them:

```
! number of layers that should be used:
levels.nz                  :   12
! layer combination, empty for no combination:
metlevel.combine           :   1 1 1 2 2 2 2 2 2 2 2 2
```

The rest of the atmoshpere (above the layers simulated by the model) is filled with concentrations from the global boundary conditions; these are used for top boundary conditions, and to simualte satellite observations. Specify the total number of layers, and how to coarsen them from the meteo layers using:

```
! total number of layers including "top"
levels.nz_top              :   19
! layer combination, empty for no combination:
metlevel.combine_top       :   1 1 1 2 2 2 2 2 2 2 2 2   1 2 3 3 4 4 4
```

NOTE: *the first top layer above the model layers should not be too thick! This will be filled with concentrations from the global boundary conditions, averaged over the layer. If the layer is too thick, the ozon concentrations will include part of the stratospheric ozone layer, and the inflow of ozone from the top will be too high!. So first top-layer should not be a combination of layers but consist of only 1 single layer*

Using a layer definition from a meteo model requires different settings depending on the originating model. An example of such a specific configuration is provided as:

```
rc/lotos-euros-data-meteo-cosmo-metlevel.rc
```

11 Meterological data

11.1 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 version 2.0. The new implementation is based on general routines that are able to handle data files in NetCDF format following common conventions.

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.

11.2 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 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 11.6);
- a keyword that describes the input mechanism in case the variable should be read from file(s) (see section 11.7); otherwise, a definition of how to compute the field should be present (see section 11.8).

11.3 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 11.4).

11.4 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)
```

11.5 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 11.7), 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.

11.6 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 is 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.

11.7 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 19.2 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, fmt1.tsurf | \
                      2013-01-01 00:00, 2020-12-31 23:59, fmt2.tsurf
```

That is, a list of time intervals and description keys 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, which often occure in operational meteo models.

The description key refers to settings that are used to define the file and variable name. For example for the surface temperature these settings could have the form:

```
fmt1.tsurf.name   :  /data/ECMWF/sfc/%{yyyy}/t2m_%{yyyymmdd}_3h.nc
fmt1.tsurf.var    :  long_name=2 metre temperature;var_name=t2m
```

The 'name' setting 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 'var' setting identifies the file variable, and consists of a ';' seperate list of 'key=value' pairs. The 'key' could be 'long_name' or 'standard_name' in this case it refers to a variable attribute in the netcdf file; a 'var_name' refers to the variable name itself. A 'standard_name' is prefered since this might be less subject to changes than a long name or the variable name. The first 'key=value' pair that identifies a variable in the file is used.

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
```

11.8 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.

11.9 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 11.5). 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 contains a single file format for all times, and tells the model that the input time resolution is 3-hourly and that linear interpolation should be applied in between:

```
meteo.t.input    : 2012-01-01 00:00, 2020-01-01 00:00, ecfile.t
meteo.t.tinterp  : interpolation=linear;step=3;units=hour
```

The file description looks like (see '`lotos-euros-meteo-ecwmwf.rc`' for the exact form):

```
ecfile.t.name    : /data/ml-tropo20/t_%{yyyymmdd}_3h.nc
ecfile.t.var     : long_name=Temperature
```

For the definition of the half-level pressure fields that are used for the mass-averaing we refer to the settings in '`lotos-euros-data.rc`'.

12 Boundary condition data

12.1 Introduction

Which boundary conditions should be read into the model is defined by a list of keywords:

```
le.bound.types              :   clim-isak clim-const clim-logan data
```

Each of the keywords defines a data set from which boundary conditions should be read. The tracers provided differ per set. If two sets contain data for the same tracer, the data corresponding to a keyword later in the list will replace the data read before.

The implementation and configuration differs per set. Each of the sets has an own module in the source code, for example:

```
le_bound_clim_emep.F90
le_bound_clim_isak.F90
le_bound_clim_logan.F90
le_bound_data.F90
          :
```

The most general set is 'data'. The configuration of this set allows input from time series of netCDF files. In future all boundary conditions will be read through this configuration, but for the momoment it is not used yet for 'older' climatological boundary conditions. The most important boundary conditions are those obtained from a global model (typically 3 hourly 3D fields), or a boundary run from LE (1 hourly resolution 3D fields) and these are all read using the 'data' configuration.

12.2 Configuration of 'data' boundary condition set

The processing of the 'data' boundary conditions follows a number of stages, and is illustrated in Figure 12-1.

- At the bottom, the original boundary condition data (blue) is obtained from for example a global model The time series could be irregular by resolution and format.
- This data is read from the files and interpolated in time (green). Typically the code will store two original fields (for example valid for 00:00 and 03:00) and interpolate this lineary to a target time (00:30). At this stage, the data is also regridded to the model resolution, but the original levels are still kept.
- In the next stage, also vertical regridding is applied (orange). This stage is seperate because the vertical structure of the model is time dependend, and vertical regridding could therefore not be done directly after reading.
- In the final stage the actuall boundary concentration fields (red) are formed. Eventually those concentrations are formed from a linear combination of original fields, for example a weight sum of dust modes that together form the 'fine' fraction in the model.

The next sections illustrate step by step the configuration of these stages.

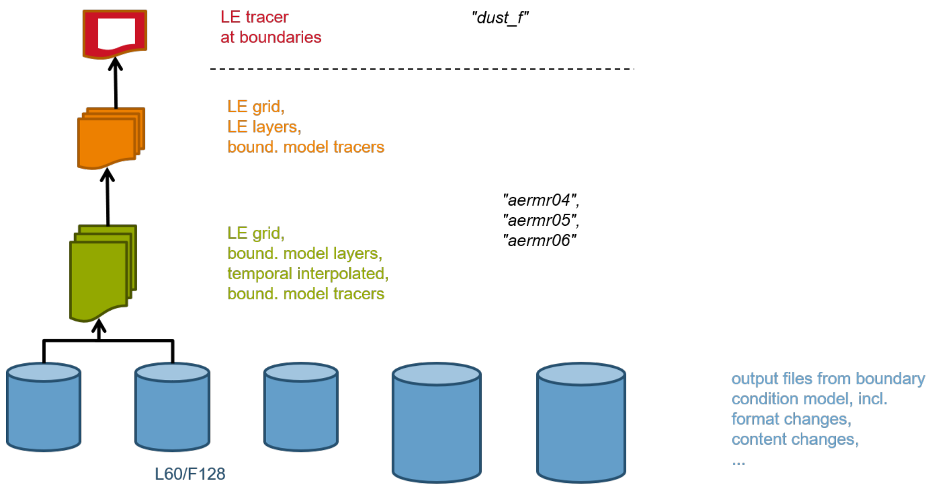
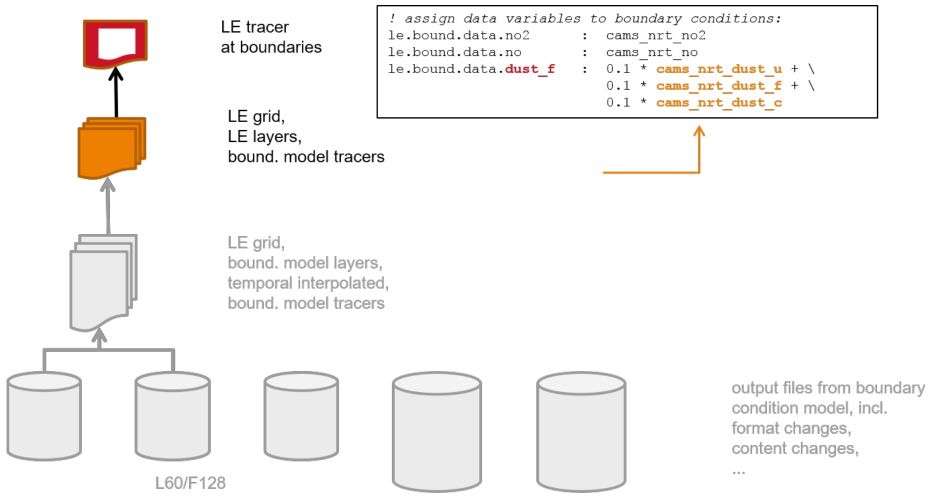


Figure 12-1 Data flow of boundary conditions.

12.2.1 *Configuration for model variables*

Figure 12-2 shows the toplevel configuration of the boundary condition data. For each of the tracers in the model, the configuration should have a definition of how to form these concentrations from variables present in the boundary condition data. The example shows for example that the fine dust fraction should use a linear combination of the original dust modes.



```
! assign data variables to boundary conditions:
le.bound.data.no2       :  cams_nrt_no2
le.bound.data.no        :  cams_nrt_no
le.bound.data.dust_f    :  0.1 * cams_nrt_dust_u + \
                           0.1 * cams_nrt_dust_f + \
                           0.1 * cams_nrt_dust_c
```

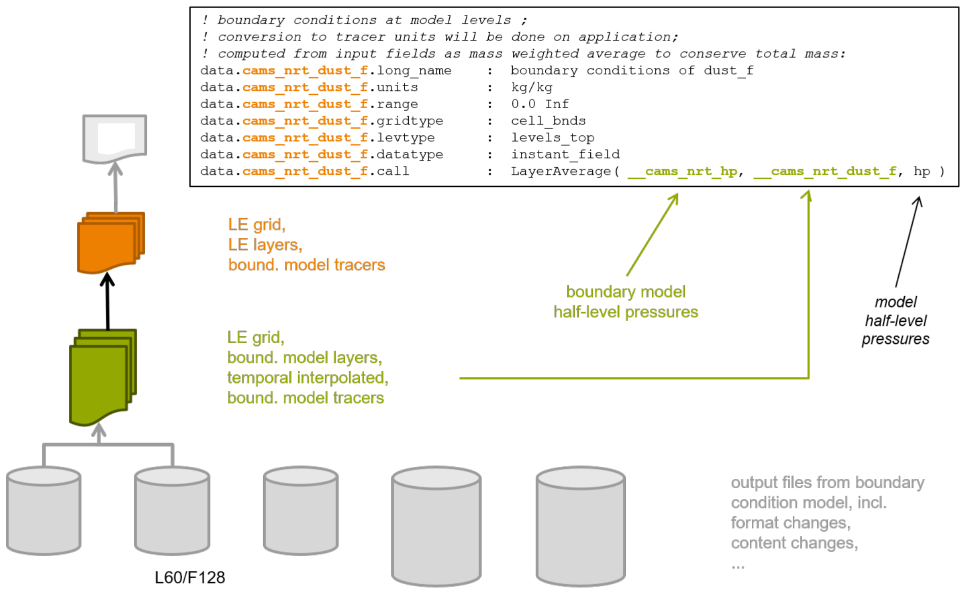
Figure 12-2 Top level configuration of boundary condition data.

12.2.2 *Configuration of boundary condition variables*

Figure 12-3 shows the configuration of the boundary condition variables. Each of these variables are stored in the 'LE_Data' structures that are also used for the meteorological data; see chapter 11 for a description.

In the example, the variable 'cams_nrt_dust_f' holds the fine dust mode read from the 'cams nrt' data set. As all boundary condition variables it holds concentrations on the model grid cells and model levels. The vertical regridding is defined using a 'call' description that tells the model to use an average over the original layers. This requires variables holding the original half level pressures and concentrations; also these should be present as 'LE_Data' variables. Here we simply added a double underscore to the names to define variables on the original levels:

```
__cams_nrt_hp
__cams_nrt_dust_f
```

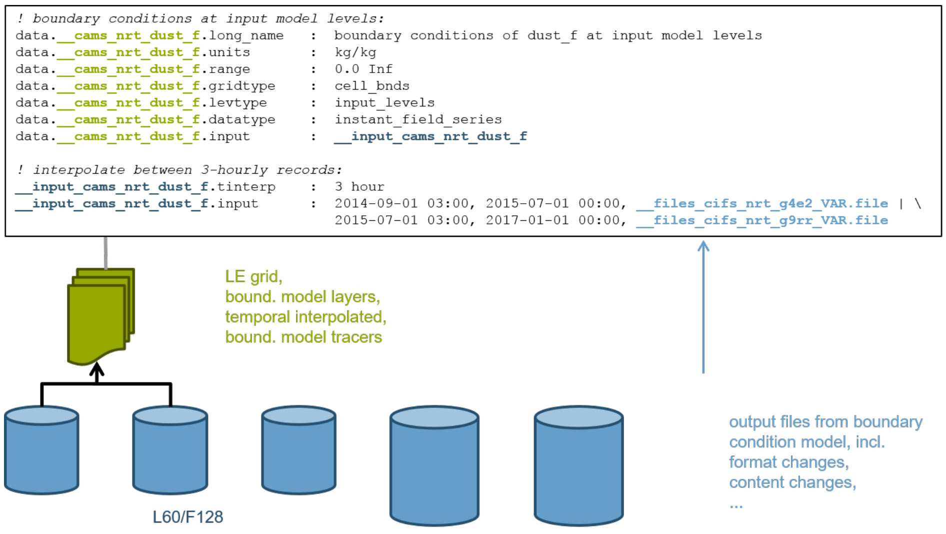


```
! boundary conditions at model levels ;
! conversion to tracer units will be done on application;
! computed from input fields as mass weighted average to conserve total mass:
data.cams_nrt_dust_f.long_name    :  boundary conditions of dust_f
data.cams_nrt_dust_f.units        :  kg/kg
data.cams_nrt_dust_f.range        :  0.0 Inf
data.cams_nrt_dust_f.gridtype     :  cell_bnds
data.cams_nrt_dust_f.levtype      :  levels_top
data.cams_nrt_dust_f.datatype     :  instant_field
data.cams_nrt_dust_f.call         :  LayerAverage( __cams_nrt_hp, __cams_nrt_dust_f, hp )
```

Figure 12-3 Configuration of regridded boundary condition variables.

12.2.3 *Configuration of boundary condition variables on original levels*

Figure 12-4 shows the configuration of the boundary condition variables on the original levels. Also these are configured as 'LE_Data' variables. In the vertical these variables are defined on input levels. The variables should be read from input files; an 'input' keyword is used for the configuration of a series of input files.



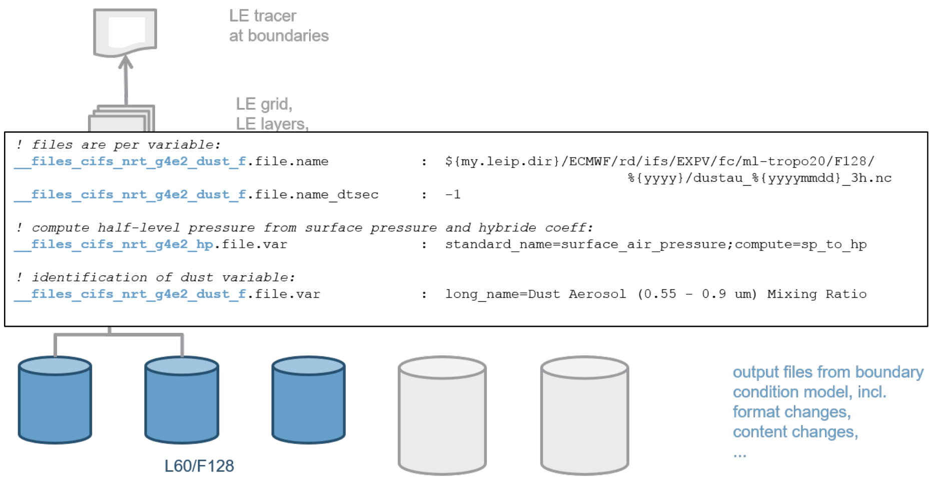
```
! boundary conditions at input model levels:
data.__cams_nrt_dust_f.long_name    :  boundary conditions of dust_f at input model levels
data.__cams_nrt_dust_f.units        :  kg/kg
data.__cams_nrt_dust_f.range        :  0.0 Inf
data.__cams_nrt_dust_f.gridtype     :  cell_bnds
data.__cams_nrt_dust_f.levtype      :  input_levels
data.__cams_nrt_dust_f.datatype     :  instant_field_series
data.__cams_nrt_dust_f.input        :  __input_cams_nrt_dust_f

! interpolate between 3-hourly records:
__input_cams_nrt_dust_f.tinterp     :  3 hour
__input_cams_nrt_dust_f.input       :  2014-09-01 03:00, 2015-07-01 00:00, __files_cifs_nrt_g4e2_VAR.file | \
                                       2015-07-01 03:00, 2017-01-01 00:00, __files_cifs_nrt_g9rr_VAR.file
```

LE grid,
bound. model layers,
temporal interpolated,
bound. model tracers
L60/F128
output files from boundary condition model, incl. format changes, content changes, ...

Figure 12-4 Configuration of boundary condition variables on original levels.

12.2.4 *Configuration of boundary condition files*

The final configuration is the description of the input file names and variables to be read. Figure 12-5 shows a part of this configuration for the dust variable.



```
! files are per variable:
__files_cifs_nrt_g4e2_dust_f.file.name          :  ${my.leip.dir}/ECMWF/rd/ifs/EXPV/fc/ml-tropo20/F128/
                                                                          %{yyyy}/dustau_%{yyyymmdd}_3h.nc
__files_cifs_nrt_g4e2_dust_f.file.name_dtsec    :  -1

! compute half-level pressure from surface pressure and hybride coeff:
__files_cifs_nrt_g4e2_hp.file.var               :  standard_name=surface_air_pressure;compute=sp_to_hp

! identification of dust variable:
__files_cifs_nrt_g4e2_dust_f.file.var           :  long_name=Dust Aerosol (0.55 - 0.9 um) Mixing Ratio
```

LE tracer
at boundaries
LE grid,
LE layers,
L60/F128
output files from boundary condition model, incl. format changes, content changes, ...

Figure 12-5 Configuration of boundary condition files and variables.

13 Generation of chemistry code

13.1 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.

13.2 Tracer and chemistry properties

An important part of the configuration is done through so-called 'properties'. Table 13.1 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 13.1 Tracer and chemistry properties.

13.3 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 13.1) that are appied to this tracer.

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

13.4 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 13.1) 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.

13.5 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.

13.6 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.

14 Labeling

14.1 rc-file

To use the labeling methode switch labeling flag to True

```
!
my.with.labeling      :   True
```

Define number of labels (default labels are added automatically)

```
!
labels.nlabel  :  2
```

Define which tracer to be labelled. Note that in a group, all tracers should be labelled

```
!
labels.labelled.specs  :  ${N_tracers}  ${S_tracers}  ${unreactive_tracers...
    ...}
```

14.2 Code

In module 'SA_Labeling.F90', the labels are defined hardcoded. Subroutine 'SA_Label_Definition...
...' is use for this.

Step 1:

Define the names of the labels and define short names with maximum of 10 characters to get proper output for Grads-ctl files.
Note number of defined labels should be equal to number defined in rc-file. In the example, labels are defined for Countries (Netherlands and abroad). Also example lines for emission-sector labeling are provided.

```
!
SA_Label_Names(1)  =  'NLD'
SA_Label_Names(2)  =  'Abroad'
!SA_Label_Names(..)        =  '..'
```

Step 2:

Match incoming emissions to the label numbers. Each incoming emission must be attached to a label

Example for countries (with label on Netherlands and Abroad)

```
if  (  icountry  ==  -999  )  then
  !  undefined  countries
  def_label  =  2
else  if  (  Emis_countries(iset)%emis_country_names(icountry)  ==  'NLD'  )...
    ...  then
  def_label  =  1
else
  def_label  =  2
end  if
```

Example for sectors (with label on Industry and others)

```
if  (  Emis_Sectors(iset)%emis_sector_names(icat)  ==  'Industry'  )  then
  def_label  =  1
else
  def_label  =  2
end  if
```

14.3 Output

From the labeling module, extra outputfiles are generated. (labelled-conc-sfc, labelled-cond-3d). Properties of those files are identical to matching concentration files described in Section 15, with an extra dimension 'label' to display the concentration resulting from each predefined label.

15 Model output

15.1 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.

15.2 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 18.2).

15.2.1 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;
  - all-advected; all tracers which are advected in the model, used for nesting approach
- vertical axis:
  - model levels, including the concentration at 2.5m (which is the measurement height, denoted as surface concentrations) 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);

Template settings for output files with surface concentrations ('conc−sfc') ,3D fields at model levels ('conc−3d') ,and 3D fields used as boundary conditions ('conc−bound') are available in the output rcfile.

15.2.2 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.

15.2.3 AOD columns

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

15.2.4 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.

Template settings for output files with surface data ('meteo−2d') and with 3D fields at model levels ('meteo−3d') are available in the output rcfile.

15.2.5 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.

15.2.6 *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.

15.2.7 *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.

15.2.8 *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.

15.3 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.

15.4 Observation simulation

The standard method to produce simulations of concentrations at observation sites is to run the DIADEM post processor (section 18.1).

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.

15.5 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.

16 Restarting a run

When a run has crashed at some point due to some problem, the restart file can be used to restart the simulation at the day where it ended, without the need for 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. Thus, when the last restart file is:

```
LE_mysimulation_state_20120823_0000.nc
```

then the day and time for the simulation should be:

```
2012-08-23 00:00:00
```

Enable startup from a restart file by the setting:

```
le.restart      : T
```

and make sure that the path and runid of the restart file are defined correctly.

It is safer but not mandatory to move the existing output directory to a different path (e.g. `output2`). Be aware that the `.ctl` files for GRADS (see section 18.2) are re-initiated 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 simulation and the restarted simulation.

17 Parallelization

LOTOS-EUROS has two options to employ multiple processors: domain decomposition using MPI, and OpenMP. These two do not exclude each other, but in practice the domain decomposition is preferred because it scales better when increasing the number of processors.

When running with MPI (domain decomposition) it is also possible to assign dedicated processor to copy input files from an archive (slow) to a local drive (fast); on some systems, this could strongly decrease the run time.

17.1 Domain decomposition

With domain decomposition, the model grid is divided into a number of sub-domains, and each sub-domain is assigned to another processor (figure 17-1). The processors use the MPI library to communicate between the domains, in particular to fill the boundary arrays using simulated concentrations from other processors.

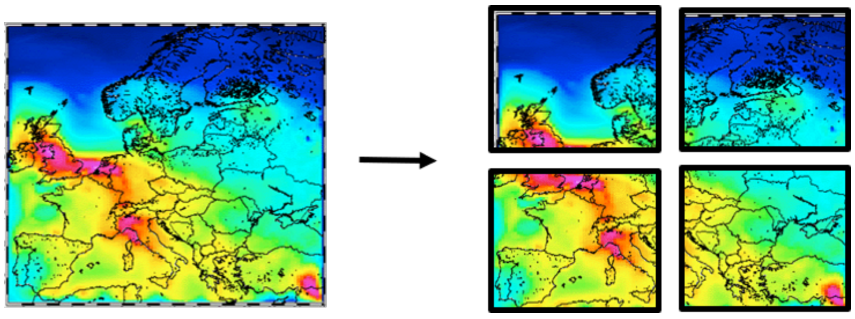


Figure 17-1 Illustration of domain decomposition.

17.1.1 Configuration

To run with domain decomposition, enable the MPI flag, specify the number of MPI tasks (sub domains), and specify how to decompose into sub-domains in x and y direction:

```
! enable MPI (True|False) ?
par.mpi                :   True

! number of mpi tasks:
par.ntask              :   4

! decomposition:
domains.x              :   2
domains.y              :   2
```

17.1.2 MPI compiler

The MPI library that is used for communication is linked with the model by using a special compiler wrapper around the native compiler. Popular MPI libraries are OpenMPI and MPICH, but also the Intel compiler suite usually comes with an MPI installation.

The name of the compiler wrapper that is used by the MPI library should be specified in the machine settings:

```
! compiler wrappers for MPI:
mpi.compiler.fc                  :   mpifort
```

17.1.3 *Logging and error messages*

Each sub domain writes standard output to a seperate log file in the output directory. The domains are numbered sequentially starting from zero, and this number is part of the per-sub-domain log file name:

```
output/le-log-d00.out
       le-log-d01.out
                 :
```

When an error occurs, it always happens first in just one of the subdomains. The error message is therefore in just one of the log files only, since the processes handling the other sub domains are simply terminated. To see which processor received the error, check the content of the error file in the run directory:

```
---[lotos-euros_run.err]----------------------------------------
MPI_ABORT was invoked on rank 3 in communicator MPI_COMM_WORLD
with errorcode 1.
...
```

In this example, the error was detected on processor 3.

17.1.4 *Comparing run times*

The DIADEM post-processor (section 18.1) could be used to visualize run times and speed up for different numbers of subdomains. Figure 17-2 shows the result for an operational configuration with 400x700 grid cells. A good speedup is achieved for all process even with a large number of subdomains. Even the input receives some speedup since this also includes the regridding which becomes a small task for a processor when the domain becomes smaller.

17.2 I/O task

Reading input files (meteo, boundaries) could take a lot of time in case the file system is rather slow. On computing servers there is often a small but very fast disk available directly on the node, for example an SSD storage. When running with MPI (domain decomposition) it is possible to assign an extra processor to copy input files from an archive (slow) to the local drive (fast), which could strongly decrease the run time.

Figure 17-3 illustrates the allocation of processors to the model domains and to the i/o task. The root domain asks the i/o processor to:

- check if selected input data for today is already available on the fast disk (here `/tmp`), and if not, copy it from the archive;
- copy the data from the archive for tomorrow;
- remove the data from yesterday from the fast disk, otherwise the fast disk might run out of space.

The copy and clean-up actions are performed by a script that is called from the i/o task. The script will install the input data for tomorrow on the fast disk, while the model is still simulating today; when the model arrives at tomorrow, the input data is already fast accessible.

To enable the i/o task, enable the following flag in the top-level settings:

```
! call installation script every day?
le.io.apply                      :   T
```

Using this flag, the machine specific settings should ensure that a job allocates an extra processor, such that the total number of MPI tasks is one more than the number of requested model sub-domains. For the TNO/HPC3 settings this is already configured.

The expert settings have configurations to ensure that the required input data for a standard run (meteo, boundary conditions, fire emissions) are installed on the fast disk. Here we shortly describe the most important settings that might require fine tuning in case of a non-standard simulations. The data sets to be copied temporary to the fast storage are identified by a list of keywords:

```
! list of data sets to be installed:
le.io.install.sets             :   meteo gfas mc–bounds
```

In case of zoom run, also the boundary output from the outer simulation should be copied:

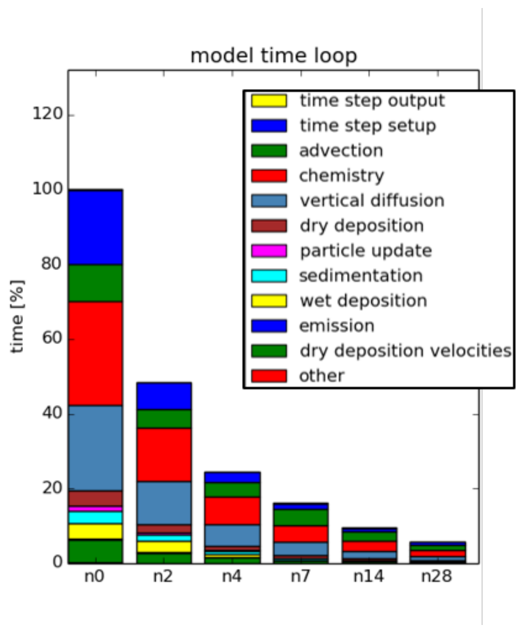


Figure 17-2 Time spent on different processes using domain decomposition with increasing numbers of subdomains.

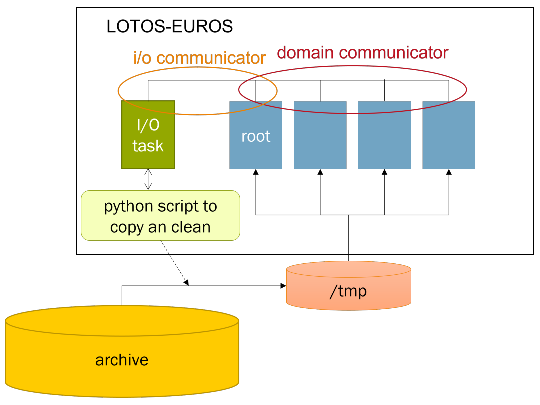


Figure 17-3 Allocation of processors for model domains and dedicated I/O task.

```
! list of data sets to be installed for zoom run:
le.io.install.sets          :  meteo gfas mc-bounds le-bounds
```

For each of these datasets, define where to copy the data from (the archive) and where to copy it to; the values are take from keys that should be defined in the machine specific settings:

```
! installation directory; empty for current:
le.io.install.meteo.dir        :  ${my.leip.dir}
! archive path with files to be installed:
le.io.install.meteo.arch       :  ${my.leip.arch}
```

Then provide a list of files that should be copied; for files including a time stamp the names should include templates for year/month/day/etc. For the meteo files this looks like:

```
! files to be installed:
le.io.install.meteo.files  : ECMWF/od/ifs/0001/an/sfc/F1280/0000/*.nc \
                             ECMWF/od/ifs/0001/fc/sfc/F1280/%Y/*_%Y%m%d_1h.nc \
                             ECMWF/od/ifs/0001/fc/L137_CL42/F1280/%Y/*_%Y%m%d_3h....
                                 ...nc
```

17.3 OpenMP

Before the introduction of the domain decomposition, the LOTOS-EUROS model used OpenMP to employ multiple processors. The domain decomposition usally scales much better with increased number of processors, but on some architectures it might still be beneficial to combine it with OpenMP.

17.3.1 Pragma's

The OpenMP standard uses special comments called *pragma's* in the source file to tell the compiler that multiple processors could be used for a certain piece of code:

```
!$OMP parallel
!$OMP   do
do j = 1, ny
  do i = 1, nx
    ! apply per grid cell:
    call Something( c(i,j,:,:), ... )
  end do  ! i
end do ! j
!$OMP   end do
!$OMP end parallel
```

17.3.2 Enable OpenMP

To run on multiple threads, enable the flag and specify the number of available threads:

```
! enable OpenMP (True,False) ?
par.openmp    :  True
! number of threads:
par.nthread   :  0
```

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.

17.3.3 *Comparing run times*

The DIADEM post-processor (section 18.1) could be used to visualize run times and speed up for different numbers of OpenMP threads. Figure 17-4 shows the result for the current version. An almost perfect speedup is achieved for the advection, vertical diffusion, and chemistry. The overall speedup is hampered however by the I/O (reading meteo, writing results) and the time step setup (emission model). With 2 threads a reasonable speedup of 1.5 is reached, while with 4 threads only a speedup of 2 is reached.

For Kalman Filter applications which perform the time loop many times but the setup only once, a much better overall speedup will be reached.

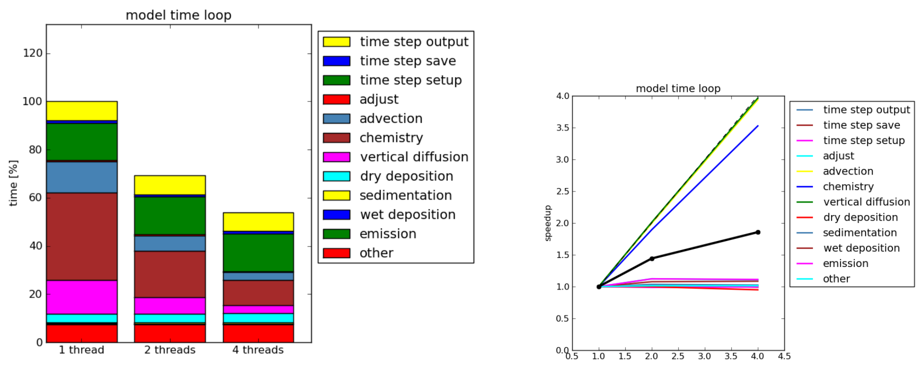


Figure 17-4 Time spent on different processes and their speedup for runs using OpenMP parallelization.

17.3.4 *Adding new OpenMP directives*

To add OpenMP directives to new code, the following approach is suggested.

1. Create a new rcfile that configures a short model run:
   - simulation over, for example, 6 hours only, since otherwise testing will take too long;
   - compiler flags 'optim−strict', since otherwise simulation results might differ slightly between two runs with and without OpenMP;
   - save restart files at least at the end of a run.
2. Use the 'diadem' post-processor (section 18.1) to show the run time profiles, produce bar graphs of the the total run time, and plots of the speedup. Search for a processes on which a relative large amount of time is spent, and is a candidate to add OpenMP. Eventually add extra timer statements to figure out in more detail which part of the model is expensive.
3. Add new OpenMP directives or change existing ones. Run 'ttb' and check if the results are still the same; if not, comment some of the new OpenMP directives and try again.
4. Repeat some the steps until the model cannot speedup anymore, or until there is are no routines left in which OpenMP directives could be inserted.

17.4 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 %)
....
```

The DIADEM post processor (section 18.1) could be used to create a browsable table of this profile bar graphs to visualize. Figure 17-5 shows an example of this for the processes in the time loop, comparing two patches.

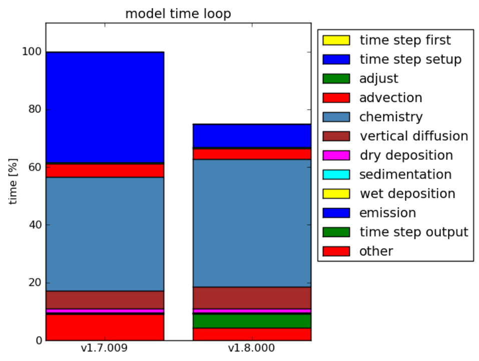


Figure 17-5 Run time spent on processes for v1.7.009 and v1.8.000 . Note that for v1.7.009 the 'time step output' task was included in the 'other' task.

18 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.

18.1 DIADEM - DIAgnostic Data End Mix

18.1.1 Introduction

The 'DIADEM' software ('*DIAgnostic Data End Mix*') is a set of tools that could be used to produce some plots for the first insight in the model results. What it can do is:

- extract statistics and other numbers from the output:
  - concentration fields averaged over the simulation time
  - time series at observation stations
  - statistics of time series
  - ...
- create plots from the extracted data
- create an html index page to the plots to be able to browse through the data

Figure 18-1 shows an example of how the main index could look like. A gallery of produced figures is shown in figure 18-2.

18.1.2 How to run

To start DIADEM, call the main script and pass an rc-file with settings to it; a template for the rc-file is also found in the DIADEM version directory:

```
<path-to-diadem-version>/bin/diadem  rc/diadem.rc
```

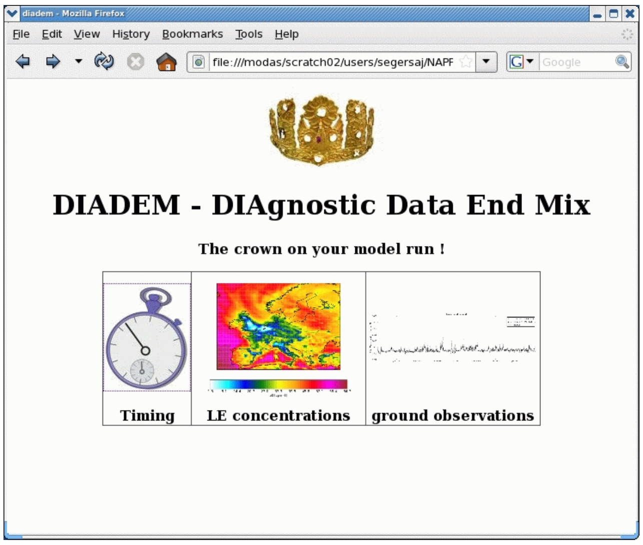
You need to change the settings to point the scripts to the location of the model output, the time range to use, which plots to make, etc.

18.2 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.

18.2.1 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.


diadem - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///modas/scratch02/users/segersaj/NAPF
Google

DIADEM - DIAgnostic Data End Mix

The crown on your model run !

Timing | LE concentrations | ground observations

Figure 18-1 Example of DIADEM main index.

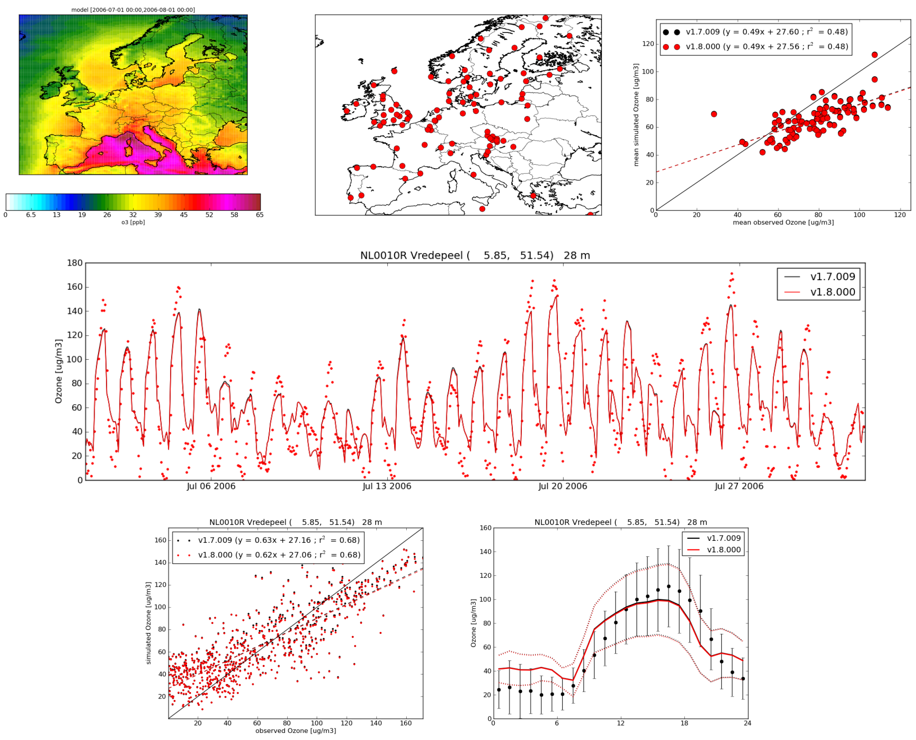


Figure 18-2 Gallery of DIADEM produced figures comparing v1.7.009 with v1.8.000 .

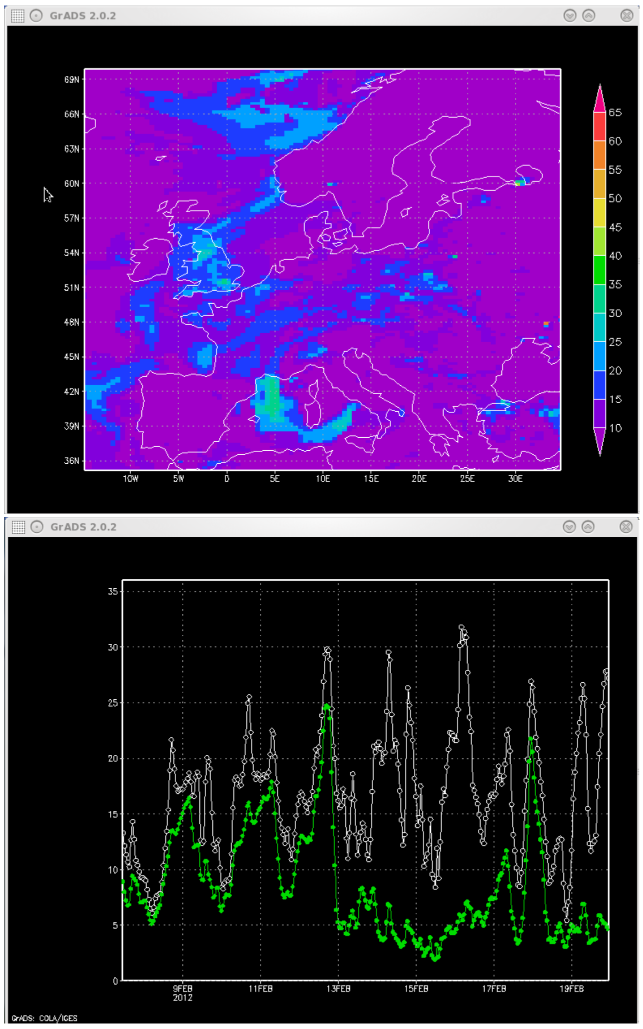


Figure 18-3 Screenshot of a GrADS graphics window with a snapshot of modelled PM10 concentrations and timeseries of PM10 and PM2.5 in gridcell corresponding to Utrecht (Netherlands)

18.2.2 Running

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

```
grads
```

19 Tools

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

After describing how to obtain a copy of the packages, the purpose and usage of each of the tools is shortly discussed. More detailed information is usually available in the tool directory itself.

19.1 DIADEM - DIAgnostic Data End Mix

DIADEM is a post-processor that could be used to create various types of plots from the model output. See section 18.1 for details.

19.2 LEIP - LOTOS-EUROS Input Processor

The LEIP scripts are used to handle the model input data. The following tasks could be performed:

- Extraction of data from ECMWF archives:
  - MARS archive with data produced by the meteorological or wave model;
  - ECFS tape archive with data produced by research models, for example the MOZART model coupled to the atmospheric model in the context of the MACC project.

  For this task LEIP needs to run on the ECMWF member state server.
- Transfer of data produced by LEIP at the member state server to a local machine.
- Collection of local available input data in order to provide it to external users.

For more information see the documentation in the package itself.

20 Creation of a new version

20.1 Versions, beta versions, and patches

For an overview of what a version and/or a patch is, see first chapter 4.

The development cycle of LOTOS-EUROS consists of the following steps:

- Release of a new *version*.
  A version number consists of a main and a subnumber, e.g. 'v2.3' . In addition the patch number could be added, which is '000' for this initial release; the total identification is then 'v2.3.000'. The code is a copy of the latest prior patch, for example 'v2.0.001'. Typically a new version is released once per year.
- Development of patches. A patch number consists of three digits, the initial patch for a new release is '000'. New patches include extra features and bug fixes, and are created when considered urgent. The creation of a new patch consists of these steps:
  - Introduction of a *beta version*. A beta version is a special project on top of an existing patch. A beta version is identified by the number that will be used for the next patch and an extension, e.g. 'v2.3.001-beta'.
  - Finishing of the beta version, including validation run to check for un-explainable changes.
  - Release of a new *patch* , e.g. 'v2.3.001', which is a copy of the latest beta version.

  After release, both the patch and the original beta version are not to be changed anymore.

The next sections contain checklists for what to do for release or creation of (beta)version or patch.

21 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 Richard Kranenburg (Dec. 2023). *LOTOS-EUROS v2.3 Reference Guide*. LOTOS-EUROS Report. TNO. URL: `www.lotos-euros.nl`.