EC-Earth 4 Getting Started Tutorial

Uwe Fladrich (SMHI)

Jan 27, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

EC-Earth4 is the next generation climate model developed by the EC-Earth consortium. This tutorial explains how to get and prepare the model source code, build the model, and how to run simple technical tests.

It is important to realise that EC-Earth4 is still at an experimental stage, and so is this tutorial. The model code, as well as the build and run-time environment will frequently change. Features will be missing and platforms may not yet be supported. An attempt is made to keep this tutorial up-to-date, which means that details will change often and without prior notice. This includes in particular URLs to repositories and references to branches, steps to build or run the model, and supported features. It is therefore advised to double check the latest on-line version of the tutorial in case of problems.

The experimental state of development also implies that some of the steps are more complicated than they ought to be. Whenever this is the case, the tutorial mentions this and improvements can be expected.

The sources for this tutorial (not the model code!) are hosted at Github and automatically published to Read the Docs on every change. If there are problems with the tutorial as such, please open an issue in the Github project or consider contributing to the tutorial with a pull request.

However, some information that complements the tutorial is not openly available, due to licensing issues (e.g. access to initial state data). This type of information is therefore kept separately at the EC-Earth Development Portal, on a Wiki page (account required). However, as much information as possible is kept open in this tutorial.

This tutorial is based on the trunk version of EC-Earth4, which, at the moment, includes OpenIFS 43r3v1 and NEMO 4.0.1. The ScriptEngine software is used to handle the build and runtime environment scripts and the OCP-Tool for automatic creation of the OASIS coupler set-up (see the *Preparations* section for details). The tutorial covers the AMIP (atmosphere-only) and GCM (atmosphere-ocean) configurations of EC-Earth4.

Note: It is possible to prepare Makefiles, configurations and run scripts manually, thus building and running EC-Earth4 without using ScriptEngine. However, this is not covered in this tutorial.

CHAPTER

TWO

PREPARATIONS

This section of the tutorial will explain how to get the EC-Earth4 source code and how to install the tools needed to build and run EC-Earth4.

Important: EC-Earth4 is not free or open source software! Some of the software components that make up EC-Earth4 require the user to aquire a license. Furthermore, the use of EC-Earth4 is restricted to EC-Earth consortium member institutions.

An account at the EC-Earth Development Portal is needed to access the EC-Earth4 source code.

2.1 Checking out the EC-Earth 4 source code

The EC-Earth4 source code is provided as a Subversion repository at the EC-Earth Development Portal. To get the current version of the code, use your login credentials and check out:

```
> svn checkout https://svn.ec-earth.org/ecearth4/trunk ece-4
```

Note: The name of the directory (`ece-4` above) is just an example. You can chose any other directory, but `ece-4` will be used throughout this tutorial.

It is assumed from now on, that you are located within the `ece-4` directory (or a subdirectory thereunder), unless noted otherwise. The current directory will be indicated by the prompt used in the code snippets.

2.2 Creating a Python virtual environment

ScriptEngine and it's dependencies are written in Python3 and they are provided as Python packages. It is usually a good idea to install packages in user space and a Python virtual environment will be used in this tutorial to install all requirements. For a detailed description of ScriptEngine installation (including other installation options like Conda), have a look at the ScriptEngine documentation

As a first step, check the Python version available by default:

```
ece-4 > python --version
Python 3.8.3
```

Make sure that the version reported is between 3.6 and 3.8 (inclusive), where the patch level (the last part of the dot-separated version number) does not matter. On some systems, the default `python` executable may still point to some Python2 version. In this case, try `python3 --version` and use `python3` also in the next step.

Now, create a new virtual environment in the EC-Earth4 directory with:

```
ece-4> python -m venv .ECE4
```

which will create a `.ECE4` directory for the environment.

Note: Again, the name of the directory for the virtual environment (`.ECE4`) is an example and any other name may be chosen. In the example above, a hidded directory (the name starting with a dot) is used in order to keep the EC-Earth4 source and runtime directory clean.

The virtual Python environment is only *created* once, but it has to be *activated* each time it is to be used, and also for each shell it is to be used in. Activating the virtual environment is done by sourcing the `activate` script in the environment directory:

```
ece-4> source .ECE4/bin/activate
```

after which the prompt should change to show the activation status:

```
(.ECE4) ece-4>
```

This prompt will be used in the examples of this tutorial, providing a reminder that the environment must be activated.

Hint: It may be convenient to create a symbolic link with `ln -s .ECE4/bin/activate`, after which the environment can be activated by typing `. activate` in the `ece-4` directory.

It is always a good idea to upgrade the Python package manager, `pip`, in a new virtual environment:

```
(.ECE4) ece-4> pip install -U pip
```

after which the environment is ready for the installation of ScriptEngine.

2.3 Installing ScriptEngine

Since ScriptEngine is provided as a package at PyPi, it can easily be installed with `pip`:

```
(.ECE4) ece-4> pip install scriptengine
```

Some of the runtime scripts use a particular ScriptEngine task package, so it is best installed right away:

```
(.ECE4) ece-4> pip install scriptengine-tasks-hpc
```

This completes the ScriptEngine installation. It can be tested with:

```
(.ECE4) ece-4> se --version
0.12.4
```

(Note that the version can differ, but it should not be lower than 0.12.4)

2.4 Installing the OCP-Tool

The OCP-Tool is used in EC-Earth4 to automatically create most of the grid description files needed by the OASIS3-MCT coupler for the combination of grids that is used for a particular experiment. The original implementation has been extended to include the grids of all EC-Earth4 components in the GCM configuration. Hence, we will have to use the extended version until the changes are merged.

The OCP-Tool is downloaded (cloned) from Github and installed locally in the EC-Earth 4 virtual environment:

```
(.ECE4) ece-4> cd ..
(.ECE4) > git clone https://github.com/uwefladrich/ocp-tool
(.ECE4) > cd ocp-tool
(.ECE4) ocp-tool> pip install -e .
```

Note: The installation of the OCP-Tools is still a bit difficult at the moment, because it is still very much under development. The last steps could be much simplified by providing an OCP-Tool package at Pypi and this will be considered in the future.

Once the installation of the OCP-Tool is successful, we can go back to the EC-Earth4 directory:

```
(.ECE4) ocp-tool> cd ../ece-4
(.ECE4) ece-4>
```

This completes the preparations of the tutorial and the EC-Earth4 is now installed along with all tools needed to build and run it.

CHAPTER

THREE

BUILDING THE EC-EARTH 4 COMPONENTS

3.1 Overview

This section will cover how the EC-Earth4 components are built (i.e. compiled). It is assumed that the EC-Earth4 source code has been checked out, ScriptEngine and the OCP-Tools are installed and the directory structure is created according to the previous section.

The source code of the EC-Earth4 components resides in subdirectory of *sources*:

```
(.ECE4) ece-4> ls -1 sources/
amip-forcing/
lpjg/
nemo-4.0.1/
oasis3-mct-4.0/
oifs-43r3/
runoff-mapper/
se/
tm5mp/
util/
xios-2.5/
```

Since this tutorial covers only the AMIP and GCM configurations, only OpenIFS (`oifs-43r3`), NEMO (`nemo-4.0.1`), OASIS (`oasis3-mct-4.0`), XIOS (`xios-2.5`), the Runoff-mapper (`runoff-mapper`) and the AMIP-Forcing-reader (`amip-forcing`) are relevant. The ScriptEngine scripts needed to built the components are located in the `se` subdirectory:

```
(.ECE4) ece-4> cd sources/se
(.ECE4) se> ls -1
platforms/
templates/
compile-all.yml
compile-amipfr.yml
compile-nemo.yml
compile-oasis.yml
compile-oifs.yml
compile-rnfm.yml
compile-xios.yml
user-settings.yml
```

The directory contains a number of YAML scripts, which are passed to ScriptEngine and will control the user and platform configuration, the preparation of makefiles and the like, and finally trigger the compilation of the components.

Detailed information about how ScriptEngine works with YAML scripts can be found in the ScriptEngine documentation. For this tutorial it suffices to mention that multiple scripts can be passed to the ScriptEngine `se` executable and are processed in order:

```
(.ECE4) se> se first.yml second.yml third.yml
2021-05-07 08:55:23 INFO [se.cli] Logging configured and started
2021-05-07 08:55:23 INFO [se.task:echo <60da064106>] Hello from firs...
Hello from first script!
2021-05-07 08:55:23 INFO [se.task:echo <fc27e03e41>] Hello from seco...
Hello from second script!
2021-05-07 08:55:23 INFO [se.task:echo <2206bc645d>] Hello from thir...
Hello from third script!
```

For a brief overview of `se` options and arguments, run `se --help`.

Note: The actual names of all scripts are just convenient names. There is nothing special about them and they could be changed at will. Scripts could also be split or merged. As far as ScriptEngine is concerned, it will just process, in order, all scripts given as arguments.

The basic approach for building the EC-Earth4 components, will be to call the scripts for user settings, the platform settings and the actual compilation commands in one go:

```
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-<COMPONENT>.yml
```

Note: It is important to always include the user and platform settings scripts in every compilation! The settings are not magically stored between the ScriptEngine runs (other than in the scripts) and must always be included!

The order in which the EC-Earth4 components must be compiled (due to inter-component dependencies) is

- OASIS
- XIOS
- OpenIFS, NEMO, AMIP-Forcing-reader, Runoff-mapper (in any order)

Hint: There is also a `compile-all.yml` script, which compiles all components in the correct order. The user and platform settings still have to be provided.

3.2 User settings

The `user-settings.yml` file is presently very simple and basically provides the path of the EC-Earth4 source code:

```yaml
# User-dependent configuration for EC-Earth 4 build
- base.getenv:
    home: HOME
- base.context:
    main:
        base_dir: "{{home}}/Projects/ece-4"
        src_dir: !noparse "{{main.base_dir}}/sources"
```

Warning: The syntax of Python, YAML, Markdown and some other modern languages and formats is based on indentation. Tab characters can not provide consistent indentation, and it has therefore become a major offence to use tab characters in source code! Make sure that your editor replaces tab characters with spaces and that you follow the indentation convention (i.e. how many spaces per level) of the source code you are working with.

The `user-settings.yml` example takes help of the `HOME` environment variable to define the EC-Earth4 directories, but that is just for convenience. Again, for details regarding the specific syntax used in the configuration definition, refer to the ScriptEngine documentation, particularly the Writing Scripts section therein.

For the course of the tutorial, it is important that the `main.base_dir` configuration parameter points to the correct directory.

3.3 Platform settings

The settings for different platforms are organised in the `platforms/` subdirectory, similar to EC-Earth3. The platform configuration is naturally a bit heavier than the user settings and comprises things like compiler and library settings, preprocessor macros and other details.

Compared to EC-Earth3, the configuration is more structured, making use of dictionary structure in YAML, For example, this is how the compilers could be configured for a particular platform:

```
- base.context:
    build:
        lang:
            fortran:
                compiler: mpif90
                flags: -g -O2 -fdefault-real-8 -fdefault-double-8 -ffree-line-length-none
                preprocessor: !noparse "{{build.lang.c.preprocessor}}"
            c:
                compiler: mpicc
                flags: -g
                preprocessor: cpp
            linker:
                command: !noparse "{{build.lang.fortran.compiler}}"
            make:
                command: gmake
```

Assuming the above configuration, the Fortran compiler could be referred to in any makefile as `{{build.lang.fortran.compiler}}` and the make command as `{{build.lang.make.command}}`.

As another example, the section to configure some needed libraries could look like:

```
- base.context:
    build:
        libs:
            mpi: null
            lapack:
                libs: [openblas]
            grib:
                base_dir: /software/sse/manual/eccodes/2.8.2/nsc1-gcc-2018a-eb
                inc_dir: !noparse "{{build.libs.grib.base_dir}}/include"
                lib_dir: !noparse "{{build.libs.grib.base_dir}}/lib"
```

(continues on next page)

(continued from previous page)

```
            libs: [eccodes_f90, eccodes, pthread]
        netcdf:
            mod_dir: /software/sse/manual/netcdf/4.4.1.1/nsc1-gcc-2018a-eb/include
            libs: [netcdff, netcdf]
        hdf5: null
```

Again, the configuration parameters are available for later use in a structured way, such as `{{build.libs.grib.lib_dir}}`.

Some details are worth further explanation in the above example: The syntax `mpi: null` and `hdf5: null` is used in YAML to denote an empty value. Thus, there is no special MPI or HDF5 configuration on this platform (configuration provided by modules).

Second, the `*.libs` parameters must be defined as YAML lists (with brackets and commas), even if there is only one library.

Hint: If you want to test user or platform setting scripts, it is possible to run these with ScriptEngine without actual compilation:

```
> se user-settings.yml platforms/<MY_PLATFORM>.yml
```

i.e. just omit the compile script(s). Thus, you can make sure the setting scripts are syntactically correct before you attempt the actual compilation.

Generally, it will be easiest to start from a known-to-work platform file, copy and adapt to the platform.

3.4 Building

It is advised to start by build the EC-Earth4 components one at a time. For the AMIP configuration, this could be done by:

```
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-oasis.yml
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-xios.yml
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-oifs.yml
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-amipfr.yml
```

and for the GCM configuration by:

```
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-oasis.yml
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-xios.yml
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-oifs.yml
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-nemo.yml
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-rnfm.yml
```

When this approach works properly, you can compile the configuration in one go, either by adding all compilation scripts at once:

```
(.ECE4) se> se user-settings.yml \
              platforms/<MY_PLATFORM>.yml \
              compile-oasis.yml \
              compile-xios.yml \
              compile-oifs.yml \
```

(continues on next page)

(continued from previous page)

```
                compile-nemo.yml \
                compile-rnfm.yml
```

or by using the convenience script `compile-all.yml`:

```
(.ECE4) se> se user-settings.yml platforms/<MY_PLATFORM>.yml compile-all.yml
```

The `compile-all.yml` script compiles *all* components (both for AMIP and GCM configurations) and allows also to configure the build process by setting `build.clean` (clean all sources and compile from scratch) and `build.jobs` (the number of parallel processes to be used):

```
- base.context:
    build:
      clean: yes
      jobs: 10
    components:
      - oasis
      - xios
      - oifs
      - nemo
      - rnfm
      - amipfr
- base.include:
    src: "compile-{{component}}.yml"
  loop:
    with: component
    in: "{{components}}"
```

Note how `compile-all.yml` is just a top-level script that is using the other `compile-*.yml` scripts under the hood.

CHAPTER

FOUR

RUNNING SIMPLE TESTS

Warning: Running even simple experiments with EC-Earth4 is a more diverse task than building the model. There are more dependencies to be fulfilled (w.r.t. user and computational environment) and more choices to be made (w.r.t. experiment configuration). Hence, the following part of the tutorial needs probably more adaptation to your needs, compared with the previous.

Furthermore, a number of choices or features may be hard-coded in the scripts, or not yet supported at all. This will change as development of EC-Earth4 and this tutorial progresses.

To prepare for a simple test experiment, we start from the ScriptEngine scripts provided in `runtime/se` and subdirectories:

```
(.ECE4) ece-4 > cd runtime/se
(.ECE4) se > ls -1
example.yml
example-slurm.yml
scriptlib/
templates/
```

The ScriptEngine runtime environment (SE RTE) is split into separate YAML scripts, partly with respect to the model component they are dealing with, and partly with respect to the runtime stage they belong to. This is done in order to provide a modular approach for different configurations and avoid overly complex scripts and duplication. Most of the YAML scripts are provided in the `scriptlib` subdirectory.

However, this splitting is not "build into" ScriptEngine or the SE RTE, it is entirely up to the user to adapt the scripts for her needs, possibly splitting up the scripts in vastly different ways.

4.1 Main structure of the run scripts

In order to use and control the modular YAML scripts provided under `scriptlib`, the user has to provide a top level run script. To make this task easier, an example script (`example.yml`) is provided. The `example.yml` script is rather simplistic. It allows for configuring of some basic experiment settings, but it does in particular not provide control over the batch job settings used for the experiment run (only the number of MPI processes per component). Hence, a slightly more complicated example script, `example-slurm.yml`, is provided, which allows for the configuration of further batch job details. However, the two example scripts share most of the configuration details explained below.

The top level run script defines a number of configuration parameters before it calls `scriptlib/main.yml` (using the ScriptEngine `base.include` task), which controls the actual model execution phases.

Exactly which configuration parameters are included in the top level script is up to the user and should be adapted to the experiment needs. The example script defines very basic settings, like the experiment ID, the experiment schedule, the model components, and a few more.

As an example, this is how the experiment ID and a description are defined:

```
base.context:
main:
    experiment_id: TEST
    experiment_description: |
        This is an example run of the ECE4 GCM
```

There could be more configuration parameters put in this script, if they are expected to change with the experiment set-up. This means that configuration parameter definitions could be moved here from `scriptlib/main.yml` (or any script called from there). For the purpose of this tutorial, this simple approach will suffice.

The top level script defines a list of components (`main.components`):

```
- base.context:
    main:
        components:
            - oifs
            - nemo
            - oasis
            - xios
            - rnfm
```

This is for the GCM configuration (OpenIFS, NEMO, OASIS, XIOS, runoff-mapper). The corresponding list for the AMIP configuration (OpenIFS, OASIS, XIOS, AMIP-Forcing-reader) would be:

```
- base.context:
    main:
        components:
            - oifs
            - oasis
            - xios
            - amipfr
```

The main structure of `scriptlib/main.yml` proceeds then as follows:

```
# Configure 'main' and all components
- base.include:
    src: "scriptlib/config-{{component}}.yml"
    ignore_not_found: yes
  loop:
    with: component
    in: "{{['main'] + main.components}}"

# On first leg: setup 'main' and all components
# ...

# Pre step for 'main' and all components
# ...

# Start model run for leg
```

(continues on next page)

(continued from previous page)

```
# ...

# Run post step for all components
# ...

# Monitoring
# ...

# Re-submit
# ...
```

Basically, the run script defines the following stages:

1. `config-*`, which sets configuration parameters for each component.
2. `setup-*`, which runs, for each component, once at the beginning of the experiment.
3. `pre-*`, which runs, for each component, at each leg before the executables.
4. `run`, which starts the actual model run (i.e. the executables).
5. `post-*`, which is run, for each component, at each leg after the model run has completed.
6. `resubmit`, which submits the model for the following leg.
7. `monitor`, which prepares data for on-line monitoring.

Not every stage has to be present in each model run, and not all stages have to be present for all components. For all stages and components that are present, there is a corresponding `scriptlib/<stage>-<component>.yml` script, which is included (via the `base.include` ScriptEngine task). Hence, the main implementation logic of `main.yml` is to go through all stages and execute all component scripts for that stage, if they exist.

Note that there is an artificial model component, called `main`, which is executed first in all stages. The corresponding `scriptlib/<stage>-main.yml` files includes tasks that are general and not associated with a particular component of the model.

4.2 Running batch jobs from ScriptEngine

ScriptEngine can send jobs to the SLURM batch system when the `scriptengine-tasks-hpc` package is installed, as described in the *Preparations* section. Here is an example of the `hpc.slurm.sbatch` task in `example.yml`:

```
# Submit batch job
- hpc.slurm.sbatch:
    account: my_slurm_account
    nodes: 14
    time: !noparse 0:30:00
    job-name: "ece-4-{{main.experiment_id}}"
    output: job.out
    error: job.out
```

What this task does is to run the entire `se` command, including all scripts given at the command line, as a batch job with the given arguments (e.g. account, number of nodes, and so on).

As a simplified example, a ScriptEngine script such as:

```
- hpc.slurm.sbatch:
    account: my_slurm_account
    nodes: 1
    time: !noparse 0:30:00
- base.echo:
    msg: Hello from batch job!
```

would in the first place submit a batch job and then stop. When the batch job executes, the first task (`hpc.slurm.sbatch`) would execute again, but do nothing because it already runs in a batch job. Then, the next task (`base.echo`) would be executed, writing the message to standard output in the batch job.

Important: The `scriptengine-tasks-hpc` package provides only support for SLURM at the moment. Support for the PBS scheduler is envisaged, but hasn't been implemented due to some peculiarities of the `qsub` command. Since SLURM is the prevalent job scheduler on most HPC systems (with the noteable exception of the ECMWF cca/b systems), it is at the moment unclear if PBS support can be prioritised any time soon. For any input on this issue, please check out related issues at the scriptengine-tasks-hpc Github repository.

4.3 The experiment schedule

ScriptEngine supports recurrence rules (rrules, RFC 5545) via the Python rrule module in order to define schedules with recurring events.

This is used in the SE RTE to specify the experiment schedule, with start date, leg restart dates, and end date. This allows a great deal of flexibility when defining the experiment, allowing for irregular legs with restarts at almost any combination of dates.

> **Warning:** Event though rrules provide a lot of flexibility for the experiment schedule, it is not certain that all parts of the SE RTE and the model code can deal with arbitrary start/restart dates. This feature is provided in order to not limit the definition of a schedule at a technical level in the RTE.

A simple schedule with yearly restarts could look like:

```
- base.context:
    schedule:
        all: !rrule >
            DTSTART:19900101
            RRULE:FREQ=YEARLY;UNTIL=20000101
```

which would define the start date of the experiment as 1990-01-01 00:00 and yearly restart on the 1st of January until the end date 2000-01-01 00:00 is reached, i.e. 10 legs.

As another example, two-year legs from 1850 until 1950 would be defined as:

```
- base.context:
    schedule:
        all: !rrule >
            DTSTART:18500101
            RRULE:FREQ=YEARLY;INTERVAL=2;UNTIL=19500101
```

4.4 The run.sh template

The start of the model component executables in the appropriate MPI environment is handled via a short shell script that is produced from a template. This happens in the scriptlib/setup-main.yml script:

```
- base.template:
    src: run-gcc+ompi.sh.j2
    dst: run.sh
```

which picks the given run script template (run-gcc+openmpi.sh.j2 in this case) from the templates/ directory, runs it through Jinja2, and places the resulting script under the name run.sh in the run directory. From there, it is later started in the "run" stage by scriptlib/run.yml:

```
- base.command:
    name: sh
    args: [run.sh]
```

There are, at the moment, a number of platform dependencies hidden in the run script template and the whole process is still under development in order to provide a robust and portable mechanism to start the MPI processes. One idea is to support starting MPI processes directly from a ScriptEngine task in scriptengine-tasks-hpc.

4.5 Initial data

The directory with initial data for EC-Earth4 is configured in example.yml:

```
- base.context:
      main:
          # ...
          inidir:  /path/to/your/initial-data
```

For now, the set if initial data can be downloaded from the SMHI Publisher at NSC, the link is given on the EC-Earth4 Tutorial Wiki page at the EC-Earth Development Portal.

Note: An account is needed to access the EC-Earth Development Portal, because the information is restricted to EC-Earth consortium member institutes.

4.6 Minimal set of changes

In the simplest case, only few things have to be changed in example.yml (or the corresponding top level script provided by the user) in order to run a simple experiment:

- main.inidir
- nemo.initial_state
- main.schedule
- possibly adaptations in the run script template
- batch job details in the batch submission task

Once all changes are made, a run can be started by:

```
(.ECE4) se > se example.yml
```

4.7 Changing the OpenIFS grid

Using the extended version of the OCP-Tools (see *Installing the OCP-Tool*) allows to select one of the predefined grids in `example.yml`:

```
- base.context:
      oifs:
          select_grid: !noparse_jinja "{{oifs.grids.TCO95L91}}"
```

The list of supported grids can be found in `scriptlib/config-oifs.yml`, together with the corresponding number of vertical levels and time steps. Note that the time step settings for different grids have not yet been thoroughly tested, and relies on ECMWF recommendations.

The OpenIFS grid can be chosen for both AMIP and GCM configurations, the OCP-Tool extensions will set up the correct combination of OpenIFS, NEMO, runoff-mapper, or AMIP-forcing-reader grids, as appropriate.

However, note that the remapping weights for OASIS3-MCT are computed at the start-up of the first leg! This will take some extra computing time, particularly for higher resolution. Work is ongoing to configure the weight computation in a way that allows parallelisation, thereby reducing the substantial overhead.

Warning: When choosing the OpenIFS grid, the time step is solely based on the ECMWF recommendations. No automatic adaptation of the time step to the coupling time step is done for GCM configurations! This means that if OpenIFS, NEMO and coupling time step are not consistent, the model with most likely crash.

CHAPTER

FIVE

SIMPLE DATA ANALYSIS TECHNIQUES

The major technical difference in the EC-Earth4 output, compared to EC-Earth3, is in the output files from the atmosphere. In contrast to EC-Earth3, OpenIFS writes output via XIOS, the XML-I/O server. The implication of this is that the output is no longer configured via *ppt* files and that the output files will be in NetCDF format and not GRIB.

The resulting NetCDF files with atmosphere data will technically contain an unstructured grid, representing the reduced Gaussian grid used by OpenIFS. Depending on the grid type (Tco or Tl) and horizontal resolution, the grids will be of different dimensions and cells will map differently to the earth surface.

5.1 Processing atmosphere data with CDO

CDO can remap grids from reduced to regular Gaussian grids with the `setgridtype,regular` command. However, the netCDF files produced by XIOS cannot be processed directly by CDO because they are supposedly unstructured. The trick is to add a proper grid description before applying the `setgridtype` command. There are 2 ways to achieve this:

```
(.ECE4) > cdo -L setgridtype,regular -setgrid,<OIFS_INIGG_FILE> output.nc regular_output.
↪nc
```

where <OIFS_INIGG_FILE> is the ICMGG*INIT file from your simulation. Don't worry that the initial file is in GRIB, CDO will handle that.

If you have to process many files it could be helpful to prepare a grid description file instead of reading the same GRIB file every time you process an output file. This grid description file is created with

```
(.ECE4) > cdo griddes <OIFS_INIGG_FILE> > griddes.txt
```

Then we add this grid description on the fly to process the `output.nc` file:

```
(.ECE4) > cdo -L setgridtype,regular -setgrid,griddes.txt output.nc regular_output.nc
```

The grid description file can be re-used every time you process output on the same grid.

The `-L` flag added in either method is optional but helps avoiding I/O errors that frequently occur with netCDF4/HDF5 files.

The above method works with linear reduced as well as with cubic orthogonal grids, all the grid information is in the ICMGG*INIT file. For spectral output (spherical harmonics) we need to distinguish between the Tl and Tco case. To process specral fields on the Tl grid one would use

```
(.ECE4) > cdo -f nc4c -z zip -L sp2gp,linear output regular_output.nc
```

and for fields on the Tco grid

```
(.ECE4) > cdo -f nc4c -z zip -L sp2gp,cubic output regular_output.nc
```

Note that older versions of CDO had `sp2gpl` which is just a short version of `sp2gp,linear`, the short form is obsolete and will disappear in the future.

5.2 Processing atmosphere data with Iris

The Python Iris package can be used to process OpenIFS data from EC-Earth4. A simple example is presented here how to load OpenIFS data, make a simple mean calculation, and plot.

The following Python packages are needed for this analysis (the `pathlib` package is used for convenience):

```
import iris
import matplotlib.pyplot as plt
from pathlib import Path
```

A function is defined for loading one or more variables from a given output file, possibly removing coordinates (see below) and applying an Iris constraint.

```
def load_oifs_data(path, varname, remove_coords=None, new_bounds=None, extract=None):
    cube = iris.load_cube(str(path), varname)
    if remove_coords:
        for c in remove_coords:
            cube.remove_coord(cube.coord(var_name=c))
    if extract:
        cube = cube.extract(extract)
    if new_bounds:
        for c in new_bounds:
            cube.coord(var_name=c).bounds = None
            cube.coord(var_name=c).guess_bounds()
    return cube
```

Another function helps to plot one or more Iris cubes:

```
def plot_cubes(*cubes, figsize=(14.5, 4.8)):  # default figsize good for 1x2 plots
    _, axes = plt.subplots(1, len(cubes), figsize=figsize)
    for c, a in zip(cubes, axes if len(cubes)>1 else [axes]):
        lons = c.coord('longitude').points
        lats = c.coord('latitude').points
        s = a.scatter(lons, lats, c=c.data, s=0.7)
        a.legend(*s.legend_elements(num=5), title=c.name())
        a.grid(True)
```

Some variables for the EC-Earth4 experiment id and the model output path for OpenIFS (first leg):

```
expid = 'TUT1'  # ECE4 experiment id
output_dir = Path(f'../run/{expid}/output/001/oifs')
```

Finally, the data is loaded and plotted. In this example, we load load mean sea level pressure (`msl`) and (`2t`) from the monthly output:

```
msl_monthly = load_oifs_data(
    output_dir / f'{expid}_monthly.nc',
    'msl',
    remove_coords=['time_centered'],
    extract=iris.Constraint(time=lambda cell: cell.point.month==2),  # February
)
tas_monthly = load_oifs_data(
    output_dir / f'{expid}_monthly.nc',
    '2t',
    remove_coords=['time_centered'],
    extract=iris.Constraint(time=lambda cell: cell.point.month==2),  # February
)
plot_cubes(msl_monthly, tas_monthly)
```

As a simple analysis, the mean sea level pressure is loaded from the daily file as well and the monthly mean is computed and compared to the earlier result:

```
msl_1d_avg = load_oifs_data(
    output_dir / f'{expid}_daily.nc',
    'msl',
    remove_coords=['time_centered'],
    new_bounds=['time_counter'],
    extract=iris.Constraint(time=lambda cell: cell.point.month==2),
).collapsed('time', iris.analysis.MEAN)
diff = msl_1d_avg - msl_monthly
plot_cubes(msl_1d_avg, diff)
print(
    f'Difference: min: {diff.data.min()}, mean: {diff.data.mean()}, max: {diff.data.
↪max()}'
)
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search