ESM Tools r6.65 UserManual

Dirk Barbi, Nadine Wieters, Paul Gierz, Fatemeh Chegini, Miguel Andrés-Martínez, Deniz Ural

Jul 02, 2026

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Welcome to the user documentation of the *ESM-Tools* software.

ESM-Tools is a collection of command-line tools to download and build different simulation models for the Earth system, such as atmosphere, ocean, biogeochemistry, hydrology, sea-ice and ice-sheet models, as well as coupled Earth System Models (ESMs). It also includes the functionality to write unified runscripts to carry out model simulations for different model setups (standalone and ESMs) on different High Performance Computing (HPC) systems.

ESM-Tools consists of the following command line tools:

Tool	Description	Documentation
esm_tools	Basic command to return information about *ESM-Tools* and create config files.	*ESM Tools*
esm_master	Command to download and build a model or ESM	*ESM Master*
esm_runscripts	Command to execute a simulation	*ESM Runscripts*

A short roadmap on how to run a model or ESM with *ESM-Tools* is given in the section *Ten Steps to a Running Model*.

ESM-Tools strongly seperates back-end code and the configuration of model or machine dependent information. A description about the syntax of these configuation files is given in the section *YAML File Syntax*.

A list of supported model components are given in the section *Supported Models*.

You are very welcome to contribute to this document. Please find more information on how to contribute here and contact the authors for feedback.

CHAPTER

TWO

GET ESM-TOOLS

2.1 Downloading

ESM-Tools is hosted on https://github.com/esm-tools/esm_tools. To get access to the software you need to be able to log into https://github.com.

Then you can start by cloning the repository `esm_tools.git`. Type into a terminal of the HPC or computer of your choice the following command line:

```
$ git clone https://github.com/esm-tools/esm_tools.git
```

This will create a local repository of *ESM-Tools* in a folder named `esm_tools`. This repository includes a collection of *yaml* configuration files containing all the information on models, coupled setups, machines, etc. in the subfolder `config`, default namelists in the folder `namelists`, example runscripts for a large number of models on different HPC systems in subfolder `runscripts`, and this documention in `docs`. Also you will find the installer `install.sh` used to install the python packages.

More information on the installation of *ESM-Tools* can be found in the section *Installation* or also in the section *Ten Steps to a Running Model*.

2.2 Accessing components in DKRZ server

Some of the *ESM-Tools* components are hosted in the gitlab.dkrz.de servers. To be able to reach these components you will need:

1. A DKRZ account (https://www.dkrz.de/up/my-dkrz/getting-started/account/DKRZ-user-account).
2. Request access to the repository from the corresponding author of the component if necessary. Feel free to contact us if you don't know who the model developers are or check the *Supported Models* section.

CHAPTER

THREE

BEFORE YOU CONTINUE

You will need python version 3.7 or later, a version of git that is not ancient (everything newer than 2.10 should be good), and up-to-date pip (`pip install -U pip`) to install the *ESM-Tools*. That means that on the supported machines, you could for example use the following settings:

machine	Settings
albedo	`$ module load git` `$ module load python`
levante.dkrz.de	`$ module load git` `$ module load python3`
glogin.hlrn.de/blogin.hlrn.de	`$ module load git` `$ module load anaconda3`

juwels.fz-juelich.de	`$ module load Stages/2022`
	`$ module load git`
	`$ module load Python/3.9.6`

aleph	`$ module load git` `$ module load python`

Note: Note that some machines might raise an error `conflict netcdf_c` when loading `anaconda3`. In that case you will need to swap `netcdf_c` with `anaconda3`:

```
$ module unload netcdf_c
$ module load anaconda3
```

CHAPTER

FOUR

TEN STEPS TO A RUNNING MODEL

1. Make sure you have git installed with version newer than 2.13, that the python version is 3.6 or later (see also *Before you continue*), and that pip is up-to-date (`pip install -U pip`). Also make sure that the location to which the python binaries will be installed (which is `~/.local/bin` by default) is in your `PATH`. For that purpose, add the following lines to one of your login or profile files, i.e. `~/.bash_profile`, `~/.bashrc`, `~/.profile`, etc.:

```
$ export PATH=$PATH:~/.local/bin
$ export LC_ALL=en_US.UTF-8
$ export LANG=en_US.UTF-8
```

2. Make sure you have a GitHub account and check our GitHub repository (`https://github.com/esm-tools/esm_tools`).

3. Download the git repository `esm_tools.git` from GitHub:

```
$ git clone https://github.com/esm-tools/esm_tools.git
```

4. In the new folder `esm_tools`, run the installer:

```
$ cd esm_tools
$ ./install.sh
```

   This should install the python packages of *ESM-Tools*. If you wonder where they end up, take a look at `~/.local/lib/python%versionnumber%/site-packages`.

5. Run `esm_master` once. You should see a long list of available targets if everything works.

6. Go to the toplevel folder into which you want to install your model codes, and run `esm_master install-`, followed by the name and the version of the model you want to install. As an example, if we want to install FESOM2:

```
$ cd /some/folder/you/wish/to/work/in
$ mkdir model_codes
$ cd model_codes
$ esm_master install-fesom-2.0
```

   You will be asked for your password to the repository of the model you are trying to install. If you don't have access to that repo yet, `esm_master` will not be able to install the model; you will have to contact the model developers to be granted access. A list of supported model can be found in the section *Supported Models*. Feel free to contact us if you don't know who the model developers are.

7. Check if the installation process worked; if so, you should find the model executable in the subfolder `bin` of the model folder. E.g.:

```
$ ls fesom-2.0/bin
```

8. Go back to the esm_tools folder, and pick a sample runscript from the runscripts subfolder. These examples are very short and can be easily adapted. Pick one that is for the model you want to run, and maybe already adapted to the HPC system you are working on. Make sure to adapt the paths to your personal settings, e.g. model_dir, base_dir etc.:

```
$ cd <PATH TO ESM TOOLS>/esm_tools/runscripts/fesom2
$ (your_favourite_editor) fesom2-ollie-initial-monthly.yaml
```

   Notice that the examples exist with the endings .yaml.

9. Run a check of the simulation to see if all needed files are found, and everything works as expected:

```
$ esm_runscripts fesom2-ollie-initial-monthly.yaml -e my_first_test -c
```

   The command line option -c specifies that this is a check run, which means that all the preparations, file system operations, ... are performed as for a normal simulation, but then the simulation will stop before actually submitting itself to the compute nodes and executing the experiment. You will see a ton of output on the screen that you should check for correctness before continuing, this includes:

   - information about missing files that could not be copied to the experiment folder
   - namelists that will be used during the run
   - the miniature .run script that is submitted the compute nodes, which also shows the environment that will be used

   You can also check directly if the job folder looks like expected. You can find it at $BASE_DIR/$EXP_ID/run_xxxxxxxxxxx, where BASE_DIR was set in your runscript, EXP_ID (probably) on the command line, and run_xxxxxxxxxxxx stands for the first chunk of your chain job. You can check the work folder, which is located at $BASE_DIR/$EXP_ID/run_xxxxxxxxxxxx/work, as well as the complete configuration used to generate the simulation, located at $BASE_DIR/$EXP_ID/run_xxxxxxxxxxxx/log.

10. Run the experiment:

```
$ esm_runscripts fesom2-ollie-initial-monthly.yaml -e my_first_test
```

That should really be it. Good luck!

CHAPTER

FIVE

INSTALLATION

5.1 Prerequisite

Make sure you have installed or loaded a python version that is 3.7 or later but not newer than 3.10 (see also *Before you continue*). Use a pip version that is up-to-date (to update run `pip install -U pip`). Also make sure that the location to which the python binaries of *ESM-Tools* will be installed (which is `~/.local/bin` by default) is in your `PATH`. For that purpose, add the following lines to one of your login or profile files, i.e. `~/.bash_profile`, `~/.bashrc`, `~/.profile`, etc.:

```
$ export PATH=$PATH:~/.local/bin
$ export LC_ALL=en_US.UTF-8
$ export LANG=en_US.UTF-8
```

5.2 Installing in local environment

After downloading the *ESM-Tools* software (see *Get ESM-Tools*) you have a new folder named `esm_tools`. In this new folder you find the script to install *ESM-Tools* on the HPC your are running this install script on.

To change into the new folder and execute the instalation script, execute the following commands:

```
$ cd esm_tools
$ ./install.sh
```

This should install all necessary python packages of *ESM-Tools*. If you wonder where they end up, take a look at `~/.local/lib/python%versionnumber%/site-packages`.

To see where *ESM-Tools* are installed, run the following command:

```
$ which esm_tools
```

A possible (default) output can be `~/.local/bin/esm_tools`.

5.3 Installing in a conda environment

First create a conda environment using the provided `environment.yaml` file:

```
conda env create -f environment.yaml -n esm_tools
```

Then activate the environment:

```
conda activate esm_tools
```

Now you can install *ESM-Tools* using the provided `install.sh` script:

```
./install.sh
```

5.4 Installing using Tykky (in ecmwf-atos)

You can also install *ESM-Tools* using a `tykky` environment. *tykky* builds a conda environment within a container, so it's use is very similar to *conda*, but with the advantages and the hurdles of containers. In the ECMWF machine you are encouraged by their admins to use *tykky* for python installations. To install *ESM-Tools* in a *tykky* environment first load the module:

```
module load tykky
```

Then build the containerize environment using the `environment.yaml` distributed inside the `esm_tools` source:

```
conda-containerize new --mamba --prefix $TYKKY_PATH/esm_tools environment.yaml
```

After this, a new *tykky* environment would have been created with the name `esm_tools`. Activate that environment by running:

```
tykky activate esm_tools
```

Now, you can use the standard `install.sh` script to install *ESM-Tools*:

```
./install.sh
```

Remember to `module load tykky` and `tykky activate esm_tools` every time you make a new login into the machine so that you can use *ESM-Tools*.

5.5 Installing in an encapsulated environment using `direnv`

Based on an alternative installation procedure, that provides an *ESM-Tools* installation employing direnv (https://direnv.net/), you can now install various encapsulated versions of *ESM-Tools* alongside each other. These different installations do not impact each other's configuration. Consequently, they can coexist in peaceful harmony. In the suggested alternative installation method all configurations will reside within the base folder of a specific *ESM-Tools* version that you install. There is no dependency on configurations outside the installation directory of a specific *ESM-Tools* version, mitigating the potential for side effects if another version of *ESM-Tools* is installed in parallel. To install *ESM-Tools* as suggested here, just follow the procedure outlined below. The steps to create the installation involve preparation of `direnv`, including setting up an environment that encapsulates potentially version-specific settings, creating a dedicated directory to which a specific version of *ESM-Tools* will be installed.

To set up such an installation in an encapsulated environent, please do the followings steps:

- Download and install `direnv` (if not yet done) with the following command:

```
$ curl -sfL https://direnv.net/install.sh | bash
```

  This will install `direnv` binary locally under `~/.local/bin/direnv`.

- Create and enter a new folder that should hold the new encapsulated environment:

```
$ mkdir esm_tools_v6.1.10 #adjust version number as appropriate
$ cd esm_tools_v6.1.10/
```

- Set up `direnv` hooks according to your used shell (see also https://direnv.net/docs/hook.html). For the `bash` shell add the following line at the end of your `.bashrc` and/or `.bash_profile` file:

```
eval "$(direnv hook bash)"
```

- Create `.envrc` file:

```
$ direnv edit .
```

  This command will create the new file `.envrc` and opens it in the default editor. You can also create the file and open it in your favorate editor.

- Add the following lines to the created `.envrc` file and save it:

```
module load python3
module load git
layout python
module load gcc
```

  Please also have a look at *Before you continue* for more details about necessary modules on the different HPCs. Make sure you keep the line `layout python`.

- Allow this `.envrc` file to be used by `direnv`:

```
$ direnv allow .
```

This environment setup will be automatically applied each time you enter this folder.

It enables us now to install *ESM-Tools* within this specific environment (see also *Get ESM-Tools* and *Installing in local environment*):

```
$ git clone https://github.com/esm-tools/esm_tools.git
$ cd esm_tools
$ ./install.sh
```

Note: Please note, that all calls of *ESM-Tools* commands for this particular installed version needs to be done within the folder that holds the `direnv` environment.

5.6 Update ESM-Tools

If you installed in editable mode as described above, you can update *ESM-Tools* by using `git`:

```
$ cd esm_tools
$ git pull origin release
```

5.7 Uninstall ESM-Tools

We are sorry to see you go! You can uninstall your current *ESM-Tools* installation in two different ways depending slightly on how you installed it.

5.7.1 Using pip

Make sure you have the most recent version of `pip` available for your system:

```
$ python3 -m pip install -U pip
```

If you are using *ESM-Tools* version 6.0.0 or higher you can use the following command to uninstall all *ESM-Tools* packages:

```
$ pip uninstall [--user] esm-tools
```

The `--user` flag may be required when using `pip` if you are not uninstalling in either a virtual environment or a global install (you would need to be root in that case).

If you are using a version of *ESM-Tools* that is older than 6.0.0 use `pip` to uninstall as follows:

```
$ pip freeze | grep esm | xargs pip uninstall -y
```

5.7.2 Manually

If you have installed *ESM-Tools* with the `install.sh` script or using `pip` with `user` mode, please follow the following steps to uninstall the software manually.

- Delete the `esm_*` executables:

  ```
  $ rm -ri ~/.local/bin/esm*
  ```

- Remove the installed Python packages:

  ```
  $ rm -ri ~/.local/lib/python3.<version>/site-packages/esm*
  ```

Note that you may have a different Python version, so the second command might need to be adapted.

CHAPTER

SIX

ESM TOOLS

ESM-Tools includes also a command line tool named `esm_tools`. It can be used to interact with several parts of our software. `esm_tools` can be used with the following options and subcommands:

6.1 Options

`esm_tools` options	Description
`--version`	Shows the version of the currently used `esm-tools`.
`--help`	Prints usage information about `esm_tools`

6.2 Subcommands

`esm_tools` subcommand	Options	Arguments	Descritpion
create-new-config	`--help`, `-t` or `--type [component (default)|setup|machine]`	NAME	Opens your $EDITOR and creates a new file for NAME
test-state	`--help`		Prints the state of the last tested experiments.

6.2.1 Usage: Top level command

To show all top-level command options and subcommands:

```
$ esm_tools --help
Usage: esm_tools [OPTIONS] COMMAND [ARGS]...

Options:
  --version  Show the version and exit.
  --help     Show this message and exit.

Commands:
  create-new-config  Opens your $EDITOR and creates a new file for NAME
  test-state         Prints the state of the last tested experiments.
```

6.2.2 Usage: Getting the Version

You can get the version number of the currently used `esm_tools` installation with:

```
$ esm_tools --version
esm_tools, version 6.20.1
```

6.2.3 Usage: Checking current testing state

You can get the current state of our automatic tests with:

```
$ esm_tools test-state
```

6.2.4 Usage: Create a new configuration

You can get a pre-generated template to add a new component with:

```
$ esm_tools create-new-config <MY_NEW_CONFIG_NAME>
Creating a new component configuration for my_new_thing

...EDITOR OPENS....

Thank you! The new configuration has been saved. Please commit it (and get in touch with␣
↪the
ESM-Tools team if you need help)!
```

You can also specify if you want to create a new `setup`, `component` or `machine` configuration file by giving the option `-t` or `--type` to the subcommand:

```
$ esm_tools create-new-config --type setup <MY_NEW_SETUP_NAME>
```

Note however that there is (as of this writing) no template available for setups!

CHAPTER

SEVEN

YAML FILE SYNTAX

7.1 What Is YAML?

YAML is a structured data format oriented to human-readability. Because of this property, it is the chosen format for configuration and runscript files in *ESM-Tools* and the recommended format for runscripts (though bash runscripts are still supported). These *YAML* files are read by the *esm_parser* and then converted into a Python dictionary. The functionality of the *YAML* files is further expanded through the *esm_parser* and other *ESM-Tools* packages (i.e. calendar math through the *esm_calendar*). The idea behind the implementation of the *YAML* format in *ESM-Tools* is that the user only needs to create or edit easy-to-write *YAML* files to run a model or a coupled setup, speeding up the configuration process, avoiding bugs and complex syntax. The same should apply to developers that would like to implement their models in *ESM-Tools*: the implementation consists on the configuration of a few *YAML* files.

Warning: *Tabs* are not allowed as *yaml* indentation, and therefore, *ESM-Tools* will return an error every time a *yaml* file with *tabs* is invoked (e.g. *runscripts* and *config* files need to be *'tab-free'*).

7.1.1 YAML-Specific Syntax

The main *YAML* **elements** relevant to *ESM-Tools* are:

- **Scalars**: numbers, strings and booleans, defined by a *key* followed by `:` and a *value*, i.e.:

```
model: fesom
version: "2.0"
time_step: 1800
```

- **Lists**: a collection of elements defined by a *key* followed by `:` and an indented list of *elements* (numbers, strings or booleans) starting with `-`, i.e.:

```
namelists:
        - namelist.config
        - namelist.forcing
        - namelist.oce
```

  or a list of the same *elements* separated by `,` inside square brackets `[elem1, elem2]`:

```
namelists: [namelist.config, namelist.forcing, namelist.oce]
```

- **Dictionaries**: a collection of *scalars*, *lists* or *dictionaries* nested inside a general *key*, i.e.:

```
config_files:
        config:  config
        forcing: forcing
        ice:     ice
```

Some relevant **properties** of the YAML format are:

- Only **white spaces** can be used for indentation. **Tabs are not allowed**.
- Indentation can be used to structure information in as many levels as required, i.e. a dictionary choose_resolution that contains a list of dictionaries (T63, T31 and T127):

```
choose_resolution:
        T63:
                levels: "L47"
                time_step: 450
                [ ... ]
        T31:
                levels: "L19"
                time_step: 450
                [ ... ]
        T127:
                levels: "L47"
                time_step: 200
                [ ... ]
```

- This data can be easily imported as *Python* dictionaries, which is part of what the *esm_parser* does.
- : should always be **followed** by a *white space*.
- **Strings** can be written both **inside quotes** (key: "string" or key: 'string') **or unquoted** (key: string).
- *YAML* format is **case sensitive**.
- It is possible to add **comments** to YAML files using # before the comment (same as in Python).

7.2 ESM-Tools Extended YAML Syntax

Warning: Work in progress. This chapter might be incomplete. Red statements might be imprecise or not true.

ESM-Tools offers extended functionality of the *YAML* files through the *esm_parser*. The following subsections list the extended *ESM-Tools* syntax for *YAML* files including calendar and math operations (see *Math and Calendar Operations*). The yaml:YAML Elements section lists the *YAML* elements needed for configuration files and runscripts.

7.2.1 Sections

Every root-level key in an ESM-Tools YAML file is a **section** (short for *yaml section*). Sections group variables related to the same aspect of the configuration and trigger specific functionality for the type of component that that section is associated to. The most common sections are named after models, coupled setups, or computers, all considered also categories of components. For example:

```yaml
fesom:
    time_step: 1800
    mesh_dir: /pool/meshes/CORE2

echam:
    resolution: T63
```

`fesom` and `echam` are sections in the example above, associated to the model components `fesom` and `echam` respectively.

Different functionality is triggered for the variables nested under these sections, depending on their component's type (e.g. model, coupled-setup, computer, system, etc.).

Sections in yaml files are validated by the *esm_parser*, that raises an error if an unrecognised key is found at the root level. See this example for an invalid section (in contrast with the valid one above):

```yaml
# Wrong: time_step must be nested under a section
time_step: 1800

# Correct
fesom:
    time_step: 1800
```

Include new sections

Valid sections are derived from the existing components defined in the experiment configuration, and include:

1. the coupled-setup section (e.g. `awicm`, `foci`, `awiesm3`, `icon-fesom`, etc.)
2. model sections (e.g. `oifs`, `fesom`, `echam`, `pism`, `oasis3mct`, `icon`, etc.)
3. HPC machine sections (`computer`)
4. system sections (`general`, `dask`)

To **add a model section** include the model component into `general.valid_model_names` in one of the yaml files:

```yaml
general:
    add_valid_model_names:
        - <my_model>

<my_model>:
    [ ... ]
```

Model sections are special in that there is additional functionality that is triggered for each of them, such as possible model compilation with `esm_master`, and in `esm_runscripts` creation of directories with their names inside the experiment dirs, file operations/tidying up after the simulation chunks, etc.

If instead you just need to **add a new yaml section** to the configuration, without triggering any additional functionality for that section, use `general.other_components`:

```
general:
    other_components:
        - <my_new_section>

<my_new_section>:
    [ ... ]
```

See *Tool-Specific Elements/Variables* for details on `valid_model_names` and `other_components`.

7.2.2 Variable Calls

Variables defined in a *YAML* file can be invoked on the same file or in other files provided that the file where it is defined is read for the given operation. The syntax for calling an already defined variable is:

```
"${name_of_the_variable}"
```

Variables can be nested in sections. To define a variable using the value of another one that is nested on a section the following syntax is needed:

```
"${<section>.<variable>}"
```

When using *esm_parser*, variables in components, setups, machine files, general information, etc., are grouped under sections of respective names (i.e. `general`, `ollie`, `fesom`, `awicm`, ...). To access a variable from a different file than the one in which it is declared it is necessary to reference the file name or label as it follows:

```
"${<file_label>.<section>.<variable>}"
```

Example

Lets take as an example the variable `ini_parent_exp_id` inside the `general` section in the *FESOM-REcoM* runscript `runscripts/fesom-recom/fesom-recom-ollie-restart-daily.yaml`:

```
general:
        setup_name: fesom-recom
        [ ... ]
        ini_parent_exp_id: restart_test
        ini_restart_dir: /work/ollie/mandresm/esm_yaml_test/${ini_parent_exp_id}/restart/
        [ ... ]
```

Here we use `ini_parent_exp_id` to define part of the restart path `ini_restart_dir`. `general.ini_restart_dir` is going to be called from the *FESOM-REcoM* configuration file `configs/setups/fesom-recom/fesom-recom.yaml` to define the restart directory for *FESOM* `fesom.ini_restart_dir`:

```
[ ... ]
ini_restart_dir: "${general.ini_restart_dir}/fesom/"
[ ... ]
```

Note that this line adds the subfolder `/fesom/` to the subdirectory.

If we would like to invoke from the same runscript some of the variables defined in another file, for example the `useMPI` variable in `configs/machines/ollie.yaml`, then we would need to use:

```
a_new_variable: "${ollie.useMPI}"
```

Bare in mind that these examples will only work if both *FESOM* and *REcoM* are involved in the *ESM-Tool* task triggered and if the task is run in *Ollie* (i.e. it will work for `esm_runscripts fesom-recom-ollie-restart-daily.yaml -e <experiment_id> ...`).

7.2.3 Switches (choose_)

A *YAML* list named as `choose_<variable>` function as a *switch* that evaluates the given `variable`. The nested element *keys* inside the `choose_<variable>` act as *cases* for the switch and the *values* of this elements are only defined outside of the `choose_<variable>` if they belong to the selected `case_key`:

```
variable_1: case_key_2

choose_variable_1:
        case_key_1:
                configuration_1: value
                configuration_2: value
                [ ... ]
        case_key_2:
                configuration_1: value
                configuration_2: value
                [ ... ]
        "*":
                configuration_1: value
                configuration_2: value
                [ ... ]
```

The key `"*"` or `*` works as an *else*.

Example

An example that can better illustrate this general description is the *FESOM 2.0* resolution configuration in `<PATH>/esm_tools/configs/fesom/fesom-2.0.yaml`:

```
resolution: CORE2

choose_resolution:
        CORE2:
                nx: 126858
                mesh_dir: "${pool_dir}/meshes/mesh_CORE2_final/"
                nproc: 288
        GLOB:
                nx: 830305
```

Here we are selecting the `CORE2` as default configuration set for the `resolution` variable, but we could choose the `GLOB` configuration in another *YAML* file (i.e. a runscript), to override this default choice.

In the case in which `resolution: CORE2`, then `nx`, `mesh_dir` and `nproc` will take the values defined inside the `choose_resolution` for `CORE2` (`126858`, `runscripts/fesom-recom/fesom-recom-ollie-restart-daily.yaml`, and `288` respectively), once resolved by the *esm_parser*, at the same **nesting level** of the `choose_resolution`.

Note: `choose_versions` inside configuration files is treated in a special way by the *esm_master*. To avoid conflicts in case an additional `choose_versions` is needed, include the compilation information inside a `compile_infos` section (including the `choose_versions` switch containning compilation information). Outside of this exception, it is possible to use as many `choose_<variable>` repetitions as needed.

7.2.4 Append to an Existing List (add_)

Given an existing list `list1` or dictionary:

```
list1:
    - element1
    - element2
```

it is possible to add members to this list/dictionary by using the following syntax:

```
add_list1:
    - element3
    - element4
```

so that the variable `list1` at the end of the parsing will contain `[element1, element2, element3, element4]`. This is not only useful when you need to build the list piecewise (i.e. and expansion of a list inside a `choose_` switch) but also as the *YAML File Hierarchy* will cause repeated variables to be overwritten. Adding a nested dictionary in this way merges the `add_<dictionary>` content into the `<dictionary>` with priority to `add_<dictionary>` elements inside the same file, and following the *YAML File Hierarchy* for different files.

Properties

- It is possible to have multiple `add_` for the same variable in the same or even in different files. That means that all the elements contained in the multiple `add_` will be added to the list after the parsing.

Exceptions

Exceptions to `add_` apply only to the environment and namelist `_changes` (see yaml:Environment and Namelist Changes (``_changes``)). For variables of the type `_changes`, an `add_` is only needed if the same `_changes` block repeats inside the same file. Otherwise, the `_changes` block does not overwrite the same `_changes` block in other files, but their elements are combined.

Example

In the configuration file for *ECHAM* (`configs/components/echam/echam.yaml`) the list `input_files` is declared as:

```
[ ... ]

input_files:
    "cldoptprops": "cldoptprops"
    "janspec": "janspec"
    "jansurf": "jansurf"
    "rrtmglw": "rrtmglw"
    "rrtmgsw": "rrtmgsw"
    "tslclim": "tslclim"
    "vgratclim": "vgratclim"
    "vltclim": "vltclim"

[ ... ]
```

However different *ECHAM* scenarios require additional input files, for example the `HIST` scenario needs a `MAC-SP` element to be added and we use the `add_` functionality to do that:

```
[ ... ]
choose_scenario:
    [ ... ]
```

(continues on next page)

(continued from previous page)

```
    HIST:
        forcing_files:
            [ ... ]
        add_input_files:
            MAC-SP: MAC-SP
    [ ... ]
```

An example for the `_changes` **exception** can be also found in the same `ECHAM` configuration file. Namelist changes necessary for *ECHAM* are defined inside this file as:

```
[ ... ]

namelist_changes:
    namelist.echam:
        runctl:
            out_expname: ${general.expid}
            dt_start:
                - ${pseudo_start_date!year}
                - ${pseudo_start_date!month}
                [ ... ]
```

This changes specified here will be combined with changes in other files (i.e. `echam.namelist_changes` in the coupled setups *AWICM* or *AWIESM* configuration files), not overwritten. However, *ECHAM*'s version `6.3.05p2-concurrent_radiation` needs of further namelist changes written down in the same file inside a `choose_` block and for that we need to use the `add_` functionality:

```
[ ... ]

choose_version:
    [ ... ]
    6.3.05p2-concurrent_radiation:
        [ ... ]
        add_namelist_changes:
            namelist.echam:
                runctl:
                    npromar: "${npromar}"
                parctl:

[ ... ]
```

7.2.5 Remove Elements from a List/Dictionary (`remove_`)

It is possible to remove elements inside list or dictionaries by using the `remove_` functionality which syntax is:

```
remove_<dictionary>: [<element_to_remove1>, <element_to_remove2>, ... ]
```

or:

```
remove_<dictionary>:
        - <element_to_remove1>
        - <element_to_remove2>
        -  ...
```

You can also remove specific nested elements of a dictionary separating the *keys* for the path by `.`:

```
remove_<model>.<dictionary>.<subkey1>.<subkey2>: [<element_to_remove1>, <element_to_
↪remove2>, ... ]
```

7.2.6 Math and Calendar Operations

The following math and calendar operations are supported in *YAML* files:

Arithmetic Operations

An element of a *YAML* file can be defined as the result of the addition, subtraction, multiplication or division of variables with the format:

```
key: "$(( ${variable_1} operator ${variable_2} operator ... ${variable_n} ))"
```

The *esm_parser* supports calendar operations through *esm_calendar*. When performing calendar operations, variables that are not given in date format need to be followed by their `unit` for the resulting variable to be also in date format, i.e.:

```
runtime: $(( ${end_date} - ${time_step}seconds ))
```

`time_step` is a variable that is not given in date format, therefore, it is necessary to use `seconds` for `runtime` to be in date format. Another example is to subtract one day from the variable `end_date`:

```
$(( ${end_date} - 1days ))
```

The units available are:

Units supported by arithmetic operations	
calendar units	seconds minutes days months years

Extraction of Date Components from a Date

It is possible to extract date components from a *date variable*. The syntax for such an operation is:

```
"${variable!date_component}"
```

An example to extract the year from the `initial_time` variable:

```
yearnew: "${initial_date!syear}"
```

If `initial_date` was 2001-01-01T00:00:00, then `yearnew` would be 2001.

The date components available are:

Date components	
ssecond	Second from a given date.
sminute	Minute from a given date.
shour	Hour from a given date.
sday	Day from a given date.
smonth	Month from a given date.
syear	Year from a given date.
sdoy	Day of the year, counting from Jan. 1.

7.2.7 Globbing

Globbing allows to use * as a wildcard in filenames for restart, input and output files. With this feature files can be copied from/to the work directory whose filenames are not completely known. The syntax needed is:

```
file_list: common_pathname*common_pathname
```

Note that this also works together with the *List Loops*.

Example

The component *NEMO* produces one restart file per processor, and the part of the file name relative to the processor is not known. In order to handle copying of restart files under this circumstances, globbing is used in *NEMO*'s configuration file (`configs/components/nemo/nemo.yaml`):

```
[ ... ]

restart_in_sources:
    restart_in: ${expid}_${prevstep_formatted}_restart*_${start_date_m1!syear!smonth!
↪sday}_*.nc
restart_out_sources:
    restart_out: ${expid}_${newstep_formatted}_restart*_${end_date_m1!syear!smonth!sday}_
↪*.nc

[ ... ]
```

This will include inside the `restart_in_sources` and `restart_out_sources` lists, all the files sharing the specified common name around the position of the * symbol, following the same rules used by the Unix shell.

7.2.8 Namelist and Coupling Changes (_changes)

The functionality `_changes` is used to control namelist and coupling changes. This functionality can be used from config files, but also runscripts. If the same type of `_changes` is used both in config files and a runscript for a simulation, the dictionaries are merged following the hierarchy specified in the *YAML File Hierarchy* chapter.

Changing Namelists

It is also possible to specify namelist changes to a particular section of a namelist:

```
echam:
        namelist_changes:
                namelist.echam:
                        runctl:
                                l_orbvsop87: false
                        radctl:
                                co2vmr: 217e-6
                                ch4vmr: 540e-9
                                n2ovmr: 245e-9
                                cecc: 0.017
                                cobld: 23.8
                                clonp: -0.008
                                yr_perp: "remove_from_namelist"
```

In the example above, the *namelist.echam* file is changed in two specific chapters, first the section `runctrl` parameter `l_orbsvop87` is set to `false`, and appropriate gas values and orbital values are set in `radctl`. Note that the special entry `"remove_from_namelist` is used to delete entries. This would translate the following fortran namelist (trucated):

```
&runctl
    l_orbvsop87 = .false.
/

&radctl
    co2vmr = 0.000217
    ch4vmr = 5.4e-07
    n2ovmr = 2.45e-07
    cecc = 0.017
    cobld = 23.8
    clonp = -0.008
/
```

Note that, although we set `l_orbsvop87` to be `false`, it is translated to the namelist as a fortran boolean (`.false.`). This occurs because *ESM-Tools* "understands" that it is writing a fortan namelist and transforms the *yaml* booleans into fortran.

For more examples, check the recipe in the cookbook (*Changing Namelist Entries from the Runscript*).

Coupling changes

Coupling changes (`coupling_changes`) are typically invoked in the coupling files (`esm_tools/configs/couplings/`), executed before compilation of coupled setups, and consist of a list of shell commands to modify the configuration and make files of the components for their correct compilation for coupling.

For example, in the `fesom-1.4+echam-6.3.04p1.yaml` used in *AWICM-1.0*, `coupling_changes` lists two `sed` commands to apply the necessary changes to the `CMakeLists.txt` files for both *FESOM* and *ECHAM*:

```
components:
- echam-6.3.04p1
- fesom-1.4
```

(continues on next page)

(continued from previous page)

```
- oasis3mct-2.8
coupling_changes:
- sed -i '/FESOM_COUPLED/s/OFF/ON/g' fesom-1.4/CMakeLists.txt
- sed -i '/ECHAM6_COUPLED/s/OFF/ON/g' echam-6.3.04p1/CMakeLists.txt
```

7.2.9 Environment Configuration

For complete documentation on environment configuration, please refer to: *ESM Environment*

7.2.10 List Loops

This functionality allows for basic looping through a *YAML list*. The syntax for this is:

```
"[[list_to_loop_through-->ELEMENT_OF_THE_LIST]]"
```

where `ELEMENT_OF_THE_LIST` can be used in the same line as a variable. This is particularly useful to handle files which names contain common strings (i.e. *outdata* and *restart* files, see *File Dictionaries*).

The following example uses the list loop functionality inside the `fesom-2.0.yaml` configuration file to specify which files need to be copied from the *work* directory of runs into the general experiment *outdata* directory. The files to be copied for runs modeling a couple of months in year 2001 are `a_ice.fesom.2001.nc`, `alpha.fesom.2001.nc`, `atmice_x.fesom.2001.nc`, etc. The string `.fesom.2001.nc` is present in all files so we can use the list loop functionality together with calendar operations (*Math and Calendar Operations*) to have a cleaner and more generalized configure file. First, you need to declare the list of unshared names:

```
outputs: [a_ice,alpha,atmice_x, ... ]
```

Then, you need to declare the `outdata_sources` dictionary:

```
outdata_sources:
      "[[outputs-->OUTPUT]]": OUTPUT.fesom.${start_date!syear}.nc
```

Here, `"[[outputs-->OUTPUT]]":` provides the *keys* for this dictionary as `a_ice`, `alpha`, `atmice_x`, etc., and `OUTPUT` is later used in the *value* to construct the complete file name (`a_ice.fesom.2001.nc`, `alpha.fesom.2001.nc`, `atmice_x.fesom.2001.nc`, etc.).

Finally, `outdata_targets` dictionary can be defined to give different names to *outdata* files from different runs using *calendar operations*:

```
outdata_targets:
      "[[outputs-->OUTPUT]]": OUTPUT.fesom.${start_date!syear!smonth}.${start_date!sday}.
↪nc
```

The values for the *keys* `a_ice`, `alpha`, `atmice_x`, ..., will be `a_ice.fesom.200101.01.nc`, `alpha.fesom.200101.01.nc`, `atmice_x.fesom.200101.01.nc`, ..., for a January run, and `a_ice.fesom.200102.01.nc`, `alpha.fesom.200102.01.nc`, `atmice_x.fesom.200102.01.nc`, ..., for a February run.

7.2.11 File Dictionaries

File dictionaries are a special type of *YAML* elements that are useful to handle input, output, forcing, logging, binary and restart files among others (see *File Dictionary Types* table), and that are normally defined inside the *configuration files* of models. File dictionary's *keys* are composed by a file dictionary `type` followed by `_` and an `option`, and the *elements* consist of a list of `file_tags` as *keys* with their respective `file_paths` as *values*:

```
type_option:
      file_tag1: file_path1
      file_tag2: file_path2
```

The `file_tags` need to be consistent throughout the different `options` for files to be correctly handled by ESM-Tools. Exceptionally, `sources` files can be tagged differently but then the option `files` is required to link sources tags to general tags used by the other options (see *File Dictionary Options* table below).

File Dictionary Types

Key	Description
analysis	User's files for their own analysis tools (i.e. to be used in the pre-/postprocessing).
bin	Binary files.
config	Configure sources.
couple	Coupling files.
ignore	Files to be ignored in the copying process.
forcing	Forcing files. An example is described at the end of this section.
log	Log files.
mon	Monitoring files.
outdata	Output configuration files. A concise example is described in *List Loops*.
restart_in	Restart files to be copied from the **experiment directory** into the **run directory** (see *Experiment Directory Structure*), during the beginning of the *computing phase* (e.g. to copy restart files from the previous step into the new run folder).
restart_out	Restart files to be copied from the **run directory** into the **experiment directory** (see *Experiment Directory Structure*), during the *tidy and resubmit phase* (e.g. to copy the output restart files from a finished run into the **experiment directory** for later use the next run).
viz	Files for the visualization tool.

File Dictionary Options

Key	Description
sources	Source file paths or source file names to be copied to the target path. **Without this option no files will be handled by ESM-Tools**. If `targets` option is not defined, the files are copied into the default *target* directory with the same name as in the *source* directory. In that case, if two files have the same name they are both renamed to end in the dates corresponding to their run (`file_name.extension_YYYYMMDD_YYYYMMDD`).
files	Links the general file tags (*key*) to the *source* elements defined in `sources`. `files` **is optional**. If not present, all *source* files are copied to the *target* directory, and the *source tags* need to be the same as the ones in `in_work` and `targets`. If present, only the *source* files included in `files` will be copied (see the *ECHAM* forcing files example below).
in_work	Files inside the *work* directory of a run (`<base_dir>/<experiment_name>/run_date1_date2/work`) to be transferred to the *target* directory. This files copy to the *target* path even if they are not included inside the `files` option. `in_work` **is optional**.
targets	Paths and new names to be given to files transferred from the *sources* directory to the *target* directory. A concised example is described in *List Loops*. `targets` **is optional**.

File paths can be absolute, but most of the `type_option` combinations have a default folder assigned, so that you can choose to specify only the file name. The default folders are:

Default folders	sources	in_work	targets
bin			
config			
ignore			
forcing			
log			
outdata	`<base_dir>/<experiment_name>/run_date1_date2/work`	`<base_dir>/<experiment_name>/run_date1_date2/work`	`<base_dir>/<experiment_name>/outdata/<model>`
restart_in			
restart_out			

Example for ECHAM forcing files

The *ECHAM* configuration file (`<PATH>/configs/echam/echam.yaml`) allows for choosing different scenarios for a run. These scenarios depend on different combinations of forcing files. File sources for all cases are first stored in `echam.datasets.yaml` (a `further_reading` file) as:

```
forcing_sources:
      # sst
      "amipsst":
              "${forcing_dir}/amip/${resolution}_amipsst_@YEAR@.nc":
                      from: 1870
                      to: 2016
      "pisst": "${forcing_dir}/${resolution}${ocean_resolution}_piControl-LR_sst_1880-
↪2379.nc"

      # sic
      "amipsic":
              "${forcing_dir}/amip/${resolution}_amipsic_@YEAR@.nc":
```

(continues on next page)

(continued from previous page)

```
                    from: 1870
                    to: 2016
      "pisic": "${forcing_dir}/${resolution}${ocean_resolution}_piControl-LR_sic_1880-
↪2379.nc"

      [ ... ]
```

Here `forcing_sources` store **all the sources** necessary for all *ECHAM* scenarios, and tag them with source *keys* (`amipsst`, `pisst`, ...). Then, it is possible to choose among these source files inside the scenarios defined in `echam.yaml` using `forcing_files`:

```
choose_scenario:
      "PI-CTRL":
              forcing_files:
                    sst: pisst
                    sic: pisic
                    aerocoarse: piaerocoarse
                    aerofin: piaerofin
                    aerofarir: piaerofarir
                    ozone: piozone
      PALEO:
              forcing_files:
                    aerocoarse: piaerocoarse
                    aerofin: piaerofin
                    aerofarir: piaerofarir
                    ozone: piozone
      [ ... ]
```

This means that for a scenario `PI-CTRL` the files that are handled by ESM-Tools will be **exclusively** the ones specified inside `forcing_files`, defined in the `forcing_sources` as `pisst`, `pisic`, `piaerocoarse`, `piaerofin`, `piaerofarir` and `piozone`, and they are tagged with new general *keys* (`sst`, `sic`, ...) that are common to all scenarios. The source files not included in `forcing_files` won't be used.

File movements

Inside the file dictionaries realm, it is possible to specify the type of movement you want to carry out (among `copy`, `link` and `move`), for a specific file or file type, and for a given direction. By default all files are `copied` in all directions.

The syntax for defining a file movement for a given file type is:

```
<model>:
    file_movements:
        <file_type>:
            <direction1>: <copy/link/move>
            <direction2>: <copy/link/move>
            [ ... ]
```

where the `file_type` in one among the *File Dictionary Types*, and the `direction` one of the following ones:

Movement file directions	
`init_to_exp`	Initial files to the corresponding general folder
`exp_to_run`	From general to the corresponding run folder
`run_to_work`	From run to the work folder on that run
`work_to_run`	From the work folder to the corresponding run folder
`all_directions`	Directions not specifically defined, use this movement

It is also possible to do the same for specific files instead of for all files inside a `file_type`. The syntax logic is the same:

```
<model>:
    file_movements:
        <file_key>:
            <direction1>: <copy/link/move>
            <direction2>: <copy/link/move>
            [ ... ]
```

where `file_key` is the key you used to identify your file inside the <`file_type`>`_files`, having to add to it `_in` or `_out` if the file is a restart, in order to specify in which direction to apply this.

Movements specific to files are still compatible with the `file_type` option, and only the moves specifically defined for files in the `file_movements` will differ from those defined using the `file_type`.

Create empty folders

File dictionaries create the necessary folders that are not present in the target path when copying files. However, some times you might need to create just an empty folder, without copying any files. This can be done by including `create_folders` in one of the component sections of the desired yaml:

```
lpj_guess:
    create_folders:
        - ${work_dir}/folder1
        - ${work_dir}/folder2
```

7.2.12 Accessing Variables from the Previous Run (`prev_run`)

It is possible to use the `prev_run` dictionary, in order to access values of variables from the previous run, in the current run. The idea behind this functionality is that variables from the previous run can be called from the yaml files with a very similar syntax to the one that would be used for the current run.

The syntax for that is as follows:

```
<your_var>: ${prev_run.<path>.<to>.<the>.<var>.<in>.<the>.<previous>.<run>}
```

For example, let's assume we want to access the *time_step* from the previous run of a *FESOM* simulation and store it in a variable called *prev_time_step*:

```
prev_time_step: ${prev_run.fesom.time_step}
```

Note: Only the single previous simulation loaded

Warning: Use this feature only when there is no other way of accessing the information needed. Note that, for example, dates of the previous run are already available in the current run, under variables such as `last_start_date`, `parent_start_date`, etc.

Branchoff experiments with `prev_run`

If you use `prev_run` variables in your model configuration files, `esm_runscripts` will require that you define a `prev_run_config_file` variable in your runscript **when you try to run a branchoff experiment**. As a branchoff is a way of restarting, `esm_runscripts` needs to know which file should use to load the `prev_run` information, but (contrary to the regular restarts within the same experiment) finding that file name is a non-trivial task: being a different experiment, the datestamps and restart frequency can differ from the parent experiment to the branchoff experiment. To overcome this problem the user needs to specify the **full path** to the `finished_config.yaml` to be used on the first run of the branchoff experiment:

```
prev_run_config_file: "/<basedir>/<expid>/config/<expid>_finished_config.yaml_<DATE>-
↪<DATE>"
```

7.2.13 Error-handling and warnings

ESM-Tools provides two distinct mechanisms for handling errors and warnings:

1. **Configuration Errors/Warnings**: For validating configuration during setup
2. **Runtime Error Detection**: For monitoring model execution and log files

Configuration Errors and Warnings

The `error` and `warning` keys allow you to define validation rules (for example with `choose_` blocks) that are checked during the configuration phase of a simulation. These are useful for:

- Validating user input in runscripts
- Enforcing version requirements
- Warning about deprecated configurations
- Preventing invalid combinations of settings

The syntax for defining errors and warnings is as follows:

Syntax

```
# Basic syntax
error/warning:
    <unique_name>:  # A descriptive name for this error/warning. You can use any string␣
↪you want.
        message: "Detailed error/warning message"  # The message to be displayed when␣
↪the error/warning is triggered.
        esm_tools_version: ">/</=/!=/version_number"  # Optional version constraint for␣
↪conditional trigerring under certain versions of ESM-Tools.
        ask_user_to_continue: True/False  # Only for warnings, to ask the user to␣
↪continue or abort.
```

Note that you can nest errors and warnings inside other blocks, for example inside a `choose_` block to trigger them based on the value of a variable, and also use `add_error` and `add_warning` to add multiple errors and warnings to the final list of errors and warnings.

Example

```
recom:
    choose_scenario:
        HIST:
            [ ... ]
        PI-CTRL:
            [ ... ]
        "*":
            add_warning:
                "wrong scenario type":
                    message: "The scenario you specified (``${recom.scenario}``) is not␣
↪supported!"
                    ask_user_to_continue: True
```

If you then define `recom.scenario: hist` instead of `HIST` then you'll get the following:

```
wrong scenario type WARNING
---------------------------
Section: recom

Wrong scenario, scenario hist does not exist

? Do you want to continue (set general.ignore_config_warnings: False to avoid␣
↪quesitoning)?
```

Runtime Error Detection (`check_error`)

The `check_error` functionality monitors model output files during execution and can take specific actions when certain patterns are detected. This is useful for:

- Detecting model crashes or errors in log files
- Monitoring for specific warning messages
- Taking automated actions based on model output

Syntax

```
<component_name>:
    check_error:
        <error_pattern>:  # Text pattern to search for in log files
            file: "path/to/logfile"  # Defaults to model's stdout/stderr
            method: "warn" or "kill"  # Action to take when pattern is found
            message: "Custom error message"  # Optional custom message
            frequency: 60  # Check interval in seconds (default: 60)
```

Parameters

- `<error_pattern>`: Text or regex pattern to search for in log files
- `file`: (Optional) Path to log file to monitor (supports variables) - Special values: `"stdout"` or `"stderr"` for default model output - Can include variables like `@jobid@` which will be replaced

- `method`: Action to take when pattern is found: - `warn`: Log a warning message - `kill`: Terminate the job and log an error
- `message`: Custom message to log when pattern is found
- `frequency`: How often to check the log file (in seconds)

Example

```
echam:
    check_error:
        "ERROR":
            method: "kill"
            message: "Fatal error in ECHAM detected"
            frequency: 30
        "WARNING":
            method: "warn"
            message: "Warning detected in ECHAM output"
```

Behavior

- The monitoring runs in a background process during model execution
- Log files are checked at the specified frequency
- When a pattern is found: - For `method: warn`: Logs a warning message - For `method: kill`: Terminates the job and logs an error

Best Practices

1. Use specific patterns to avoid false positives
2. Include helpful error messages that explain the issue
3. Test error conditions to ensure they're properly detected

CHAPTER

EIGHT

TEMPLATING FILES WITH JINJA

esm_runscripts supports jinja templating. This is desirable because some times we might want that esm_runcripts "edits" some source files based on parameters defined in the configuration yamls or in its current scope.

For example, in the following domain_def.xml file:

```
<domain_definition>
<!-- Definition of the native domain of the model -->
<domain id="reduced_gaussian" long_name="reduced Gaussian grid" type="gaussian" />

<!-- Definition of regular Gaussian domains -->
<domain_group id="regular_domains" type="rectilinear" >
   <domain id="regular" long_name="regular grid" ni_glo="{{xios.ni_glo}}" nj_glo="{{xios.
↪nj_glo}}" >
      <generate_rectilinear_domain />
      <interpolate_domain order="1" write_weight="true" />
   </domain>
</domain_group>
</domain_definition>
```

we want that ni_glo and nj_glo get the values from the parameters xios.ni_glo and xios.nj_glo defined in esm_tools/configs/components/xios/xios.yaml.

This is achieved in esm_runscripts by:

1. Having a source file where it's name finishes with .j2, and that uses the syntax in the example above (including the ESM-Tools parameters to be substituted within double curly braces {{ }}), or using any other jinja syntax (read more about jinja syntax in the jinja's documentation).
2. Including that file as a source in any of the configuration yamls involved in the simulation. For example, to use the namelists/oifs/43r3/xios/domain_def.xml.j2 template to generate the domain_def.xml file, we would include it in the esm_tools/configs/components/xios/xios.yaml as follows (the same way we include any file that must be copied/moved/linked to the working directory, i.e., *File Dictionaries*):

   ```
   config_sources:
       domain_def: ${xml_dir}/domain_def.xml.j2

   config_in_work:
       domain_def: domain_def.xml
   ```

Warning: If a template's name does not end with .j2, it will be copied as is, without any substitution.

Note: As with any other file dictionary, you can ommit the target (in this case `config_in_work.domain_def`) and the file will be copied to the target directory with the same name as the source file, except that for jinja files, the `.j2` will be removed.

CHAPTER

NINE

YAML FILE HIERARCHY

9.1 Hierarchy of YAML configuration files

The following graph illustrates the hierarchy of the different YAML configuration files.

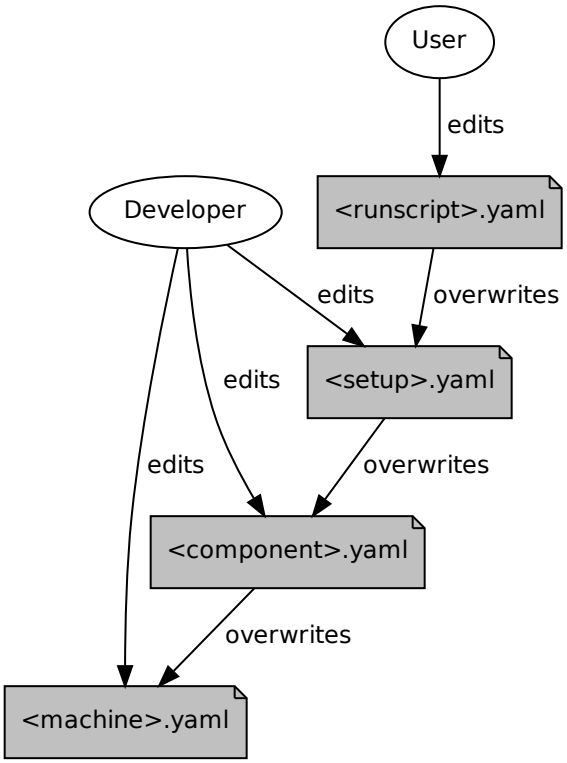


Fig. 1: ESM-Tools configuration files hierarchy

A parameter defined in, for example, a `component` will be overwritten by the same parameter if it's defined in a `setup` or the `runscript`.

9.2 How can I know where a parameter is defined?

One of the first steps in every *ESM-Tools* operation is to load the configuration files and merge their information following the hierarchy described above. During runtime, this final dictionary is stored in a yaml file under the following possible paths:

- <base_dir>/<expid>/run_<DATE>/configs/<expid>_finished_config.yaml
- <base_dir>/<expid>/configs/<expid>_finished_config.yaml

One can use this file to check the final value of the desired parameter. The same information can also be retrieved by using inspect command (*Arguments*):

```
esm_runscripts <your_runscript> -e <expid> --inspect config
```

Additionally, these files/output also contain information about the provenance of the value in the form of comments:

```
fesom:
    model: fesom  # <PATH>/esm_tools/configs/components/fesom/fesom-2.0.yaml,line:4,col:8
    branch: 2.0.2 # <PATH>/esm_tools/configs/components/fesom/fesom-2.0.yaml,line:17,
↪col:13
    version: 2 # <PATH>/esm_tools/configs/setups/awicm3/awicm3.yaml,line:399,col:18
    type: ocean # <PATH>/esm_tools/configs/components/fesom/fesom-2.0.yaml,line:7,col:7
    comp_command: mkdir -p build; cd build; cmake -DOIFS_COUPLED=ON -DFESOM_COUPLED=ON -
↪DCMAKE_INSTALL_PREFIX=../ ..;   make install -j `nproc --all` # <PATH>/esm_tools/
↪configs/setups/awicm3/awicm3.yaml,line:414,col:31
    clean_command: rm -rf build CMakeCache.txt # <PATH>/esm_tools/configs/components/
↪fesom/fesom-2.0.yaml,line:10,col:16
    required_plugins:
    - git+https://github.com/esm-tools-plugins/tar_binary_restarts  # <PATH>/esm_tools/
↪configs/components/fesom/fesom-2.0.yaml,line:13,col:3
    install_bins: bin/fesom.x  # <PATH>/esm_tools/configs/components/fesom/fesom-2.0.
↪yaml,line:22,col:19
```

CHAPTER

TEN

ESM-TOOLS VARIABLES

The *esm_parser* is used to read the multiple types of *YAML* files contained in *ESM-Tools* (i.e. model and coupling configuration files, machine configurations, runscripts, etc.). Each of these *YAML* files can contain two type of *YAML* elements/variables:

- **Tool-specific elements**: *YAML-scalars*, *lists* or *dictionaries* that include instructions and information used by *ESM-Tools*. These elements are predefined inside the *esm_parser* or other packages inside *ESM-Tools* and are used to control the *ESM-Tools* functionality.
- **Setup/model elements**: *YAML-scalars*, *lists* of *dictionaries* that contain information defined in the model/setup config files (i.e. `awicm.yaml`, `fesom.yaml`, etc.). This information is model/setup-specific and causes no effect unless it is combined with the **tool-specific elements**. For example, in `fesom.yaml` for *FESOM-1.0* the variable `asforcing` exists, however this means nothing to *ESM-Tools* by its own. In this case, this variable is used in `namelist_changes` (a tool-specific element) to state the type of forcing to be used and this is what actually makes a difference to the simulation. The advantage of having this variable already defined and called in `namelist_changes`, in the `fesom.yaml` is that the front-end user can simply change the forcing type by changing the value of `asforcing` (no need for the front-end user to use `namelist_changes`).

The following subsection lists and describes the **Tool-specific elements** used to operate *ESM-Tools*.

Note: Most of the **Tool-specific elements** can be defined in any file (i.e. *configuration file*, *runscript*, ...) and, if present in two files used by ESM-Tools at a time, the value is chosen depending on the ESM-Tools file priority/read order (*YAML File Hierarchy*). Ideally, you would like to declare as many elements as possible inside the *configuration files*, to be used by default, and change them in the *runscripts* when necessary. However, it is ultimately up to the user where to setup the Tool-specific elements.

10.1 Tool-Specific Elements/Variables

The following keys should/can be provided inside configuration files for models (`<PATH>/esm_tools/configs/components/<name>/<name>.yaml`), coupled setups (`<PATH>/esm_tools/configs/setups/<name>/<name>.yaml`) and runscripts. You can find runscript templates in `esm_tools/runscripts/templates/`).

10.1.1 Compile time variables

Key	Section	Description
execution_mode	general	Takes the value `compile` during compile time. Can be used in `choose_` blocks with `choose_general.execution_mode`.
model	general	Name of the model/setup as listed in the config files (`esm_tools/configs/components` for models and `esm_tools/configs/setups` for setups).
setup_name	general	Name of the coupled setup.
version	general	Version of the model/setup (one of the available options in the `available_versions` list).
available_versions	<component>	List of supported versions of the component or coupled setup.
git-repository	<component>	Address of the model's git repository.
branch	<component>	Branch from where to clone.
destination	<component>	Name of the folder where the model is downloaded and compiled, in a coupled setup.
comp_command	<component>	Command used to compile the component.
install_bins	<component>	Path inside the component folder, where the component is compiled by default. This path is necessary because, after compilation, ESM-Tools needs to copy the binary from this path to the <`component/setup_path`>`/bin` folder.
source_code_permissions	<component>	Sets the file permisions for the source code using `chmod <source_code_permissions> -R <source_code_folder>.

10.1.2 Run-time variables

Key	Section	Description
account	general	User account of the HPC system to be used to run the experiment.
base_dir	general	Path to the directory that will contain the experiment folder (where the experiment will be run and data will be stored).
compute_time	general	Estimated computing time for a run, used for submitting a job with the job scheduler.
conda.env	conda	Full path (or name) of a conda environment to activate in the job script. Optional: only needed to override the conda environment auto-detected from the one ESM-Tools was launched with. See *Running an experiment with conda*.
conda.root	conda	Root installation directory of conda (e.g. `/path/to/conda`), used together with `conda.env` to source `{conda.root}/bin/activate` before activating the environment. See *Running an experiment with conda*.
create_folders	<component>	List of absolute paths of the folders to be created. See *Create empty folders*.
esm_configs_dir	general	Absolute path to the ESM-Tools configs directory (`configs/`). Set automatically by `esm_parser` at startup. Use as `${general.esm_configs_dir}/...` in YAML files to reference scripts and files under the configs tree.
esm_couplings_dir	general	Absolute path to the ESM-Tools couplings directory (`couplings/`). Set automatically by `esm_parser` at startup. Use as `${general.esm_couplings_dir}/...` in YAML files to reference coupling configurations.

continues on next page

Table 1 – continued from previous page

Key	Section	Description
esm_namelist_dir	general	Absolute path to the ESM-Tools namelists directory (`namelists/`). Set automatically by `esm_parser` at startup. Use as `${general.esm_namelist_dir}/...` in YAML files to reference namelist templates.
esm_runscript_dir	general	Absolute path to the ESM-Tools runscripts directory (`runscripts/`). Set automatically by `esm_parser` at startup. Use as `${general.esm_runscript_dir}/...` in YAML files to reference runscripts or `further_readings`.
executable	<component>	Name of the component executable file, as it shows in the `<component/setup_path>/bin` after compilation.
execution_command	<component>	Command for executing the component, including `${executable}` and the necessary flags.
execution_mode	general	Takes the value `run` during run time. Can be used in `choose_` blocks with `choose_general.execution_mode`.
expid	general	ID of the experiment. This variable can also be defined when calling `esm_runscripts` with the `-e` flag.
File Dictionaries	<component>	*YAML* dictionaries used to handle input, output, forcing, logging, binary and restart files (see *File Dictionaries*).
force_overwrite_in_file_movements	general	The file movements will not overwrite existing files. Only set to `True` if you know why you would want to do that (e.g to overwrite files in a failed tidy task)."
heterogeneous_parallelization	computer	A boolean that controls whether the simulation needs to be run with or without heterogeneous parallelization. When `false` OpenMP is not used for any component, independently of the value of `omp_num_threads` defined in the components. When `true`, `open_num_threads` needs to be specified for each component using OpenMP. `heterogeneous_parallelization` variable **needs to be defined inside the** `computer` section of the runscript. See *Heterogeneous Parallelization Run (MPI/OpenMP)* for examples.
ini_restart_dir	<component>	Path of the restarted experiment in case the current experiment runs in a different directory. For this variable to have an effect `lresume` needs to be `true` (e.g. the experiment is a restart).
ini_restart_expid	<component>	ID of the restarted experiment in case the current experiment has a different `expid`. For this variable to have an effect `lresume` needs to be `true` (e.g. the experiment is a restart).
install_missing_plugins	general	A boolean to indicate whether `esm_runscripts` needs to install missing plugins (`True`, default) or not (`False`). Implemented to solve a problem with the `esm_tests` CI in GitHub where we might not have access to some repositories.

continues on next page

Table 1 – continued from previous page

Key	Section	Description
launched_with_conda	computer	Boolean set automatically to `True` when ESM-Tools is launched from inside an active conda environment, `False` otherwise. Set internally, not meant to be defined by the user. See *Running an experiment with conda*.
lresume	<component>	Boolean to indicate whether the run is an initial run or a restart.
mail_type	general/computer	Value for the SBATCH flag `--mail-type` (see https://slurm.schedmd.com/sbatch.html#OPT_mail-type)
mail_user	general/computer	Value for the SBATCH flag `--mail-user` (see https://slurm.schedmd.com/sbatch.html#OPT_mail-user)
model_dir	general/<component>	Absolute path of the model directory (where it was installed by *esm_master*).
namelists	<component>	List of namelist files required for the model.
namelist_changes	<component>	Functionality to handle changes in the namelists from the yaml files (see *Changing Namelists*).
nproc	<component>	Number of processors to use for the model.
nproca/nprocb	<component>	Number of processors for different MPI tasks/ranks. Incompatible with `nproc`.
nnodes_envvar	computer	Name of the environment variable holding the number of allocated nodes (e.g. `SLURM_JOB_NUM_NODES`).
omp_num_threads	<component>	A variable to control the number of OpenMP threads used by a component during an heterogeneous parallelization run. This variable **has to be defined inside the section of the components** for which OpenMP needs to be used. This variable will be ignored if `computer.heterogeneous_parallelization` is not set to `true`.
parallel_file_movements	general	Controls how file movements are parallelized. `"dask"` (default) distributes I/O across all compute nodes via a Dask cluster, `"threads"` uses local threads on a single node, `False` runs sequentially. See *Parallel File Movements*.
pool_dir	general	Path to the pool directory to read in mesh data, forcing files, inputs, etc.
post_processing	<component>	Boolean to indicate whether to run postprocessing or not.
post_run_commands	computer	Shell commands appended to the job script after the model execution and before resubmission. Can be a `string` or a `list` of `strings`.
pre_recipe.exclude_job_types	general	List of job types that skip `pre_recipe.steps` (default: `["prepare", "prepexp", "observe"]`).
pre_recipe.steps	general	List of recipe step names injected before the main recipe (e.g. `["initialize_dask_cluster"]`). Steps listed here run for all job types except those in `pre_recipe.exclude_job_types`.
save_batch_env_patterns	computer	List of grep patterns used to capture and restore batch system environment variables across job script stages (e.g. `["SLURM"]` or `["PBS"]`).
setup_dir	general	Absolute path of the setup directory (where it was installed by *esm_master*).
system_components	general	List of non-model config components not included in file-list iteration loops (default: `["general", "dask"]`).
other_components	general	Additional section names accepted at the root level of the assembled config (validation only — these sections do **not** participate in file operations). Extend it with `add_other_components` in a runscript or config file. See *Sections*.
valid_model_names	general	List of model components recognised at the root level of the assembled config. Models listed here trigger full file-operation and directory-creation functionality in `esm_runscripts` (experiment sub-directories, file movements, tidy-up, etc.). Extend it with `add_valid_model_names` in a runscript or config file. See *Sections*.

continues on next page

Table 1 – continued from previous page

Key	Section	Description
time_step	<component>	Time step of the component in seconds.

10.1.3 Dask variables

Variables in the `dask` section control the Dask cluster used for parallel file movements. See *Parallel File Movements* for usage details.

Key	Section	Description
actions	dask	List of actions that trigger Dask cluster initialization (default: `["parallel_file_movements"]`).
client_timeout	dask	Timeout in seconds when probing the Dask scheduler status (default: `0.05`).
init_scheduler_cmd	dask	Shell command to start the Dask scheduler. Defined per batch system (e.g. in `slurm.yaml`).
init_workers_cmd	dask	Shell command to start the Dask workers. Defined per batch system (e.g. in `slurm.yaml`).
parallel_file_movements	general	Controls how file movements are parallelized. `"dask"` (default) distributes I/O across all compute nodes via a Dask cluster, `"threads"` uses local threads on a single node, `False` runs sequentially. See *Parallel File Movements*.
poll_interval	dask	Polling interval in seconds for Dask cluster readiness checks (default: `0.5`).
scheduler_json	dask	Full path to the Dask scheduler JSON file used for client connections (default: `${general.thisrun_work_dir}/dask_scheduler.json`).
workers_timeout	dask	Maximum time in seconds to wait for Dask workers to become available (default: `5`).

10.1.4 Calendar variables

Key	Description
initial_date	Date of the beginning of the **simulation** in the format YYYY-MM-DD. If the simulation is a restart, `initial_date` marks the beginning of the restart.
final_date	Date of the end of the **simulation** in the format YYYY-MM-DD.
start_date	Date of the beginning of the **current run**.
end_date	Date of the end of the **current run**.
current_date	Current date of the run.
next_date	Next run initial date.
nyear, nmonth, nday, nhour, nminute	Number of time unit per run. They can be combined (i.e. `nyear: 1` and `nmonth: 2` implies that each run will be 1 year and 2 months long).
parent_date	Ending date of the previous run.

10.1.5 Coupling variables

Key	Description
grids	List of grids and their parameters (i.e. `name`, `nx`, `ny`, etc.).
coupling_fields	List of coupling field dictionaries containing coupling field variables.
nx	When using *oasis3mct*, used inside `grids` to define the first dimension of the grid.
ny	When using *oasis3mct*, used inside `grids` to define the second dimension of the grid.
coupling_methods	List of coupling methods and their parameters (i.e. `time_transformation`, `remapping`, etc.).
time_transformation	Time transformation used by *oasis3mct*, defined inside `coupling_methods`.
remapping	Remappings and their parameters, used by *oasis3mct*, defined inside `coupling_methods`.

10.1.6 Environment variables

Key	Section	Description
general_actions	computer	List of general shell actions to be included in the compilation and run scripts. These are added directly to the script without any prefix.
module_actions	computer	List of module actions to be included in the compilation and run scripts. Each entry will be prefixed with `module` in the generated script.
spack_actions	computer	List of Spack actions to be included in the compilation and run scripts. Each entry will be prefixed with `spack` in the generated script.
export_vars	computer	Dictionary of environment variables to be exported in the script. Each key-value pair will generate an `export KEY=VALUE` line.
unset_vars	computer	List of environment variables to be unset in the script. Each entry will generate an `unset VARIABLE` line.
include_env_from_components	computer/component	Boolean that controls whether environment variables from component files should be included. Can be set globally in the computer section or per-component. Default: `True`.
merge_component_envs	computer	Dictionary with `compile` and `run` keys that controls whether environments from all components should be merged. For `compile` the default is false (each component maintains its own environment), for `run` the default is true (environments are merged).

Note: For more detailed information on all environment configuration options, including attribute-based selection, coupled setup environment control, and advanced environment management features, please refer to the *ESM Environment* documentation.

10.1.7 Other variables

Key	Description
metadata	List to incude descriptive information about the model (i.e. `Authors`, `Institute`, `Publications`, etc.) used to produce the content of *Supported Models*. This information should be organized in nested *keys* followed by the corresponding description. Nested *keys* do not receive a special treatment meaning that you can include here any kind of information about the model. Only the *Publications key* is treated in a particular way: it can consist of a single element or a *list*, in which each element contains a link to the publication inside <> (i.e. `- Title, Authors, Journal, Year. <https://doi.org/...>`).

CHAPTER

ELEVEN

ESM MASTER

11.1 Usage: esm_master

To use the command line tool `esm_master`, just enter at a prompt:

```
$ esm_master
```

The tool may ask you to configure your settings; which are stored in your home folder under `${HOME}/.esmtoolsrc`. A list of avaiable models, coupled setups, and available operations are printed to the screen, e.g.:

```
setups:
   awicm:
      1.0: ['comp', 'clean', 'get', 'update', 'status', 'log', 'install', 'recomp']
      CMIP6: ['comp', 'clean', 'get', 'update', 'status', 'log', 'install', 'recomp']
      2.0: ['comp', 'clean', 'get', 'update', 'status', 'log', 'install', 'recomp']
[...]
```

As can be seen in this example, `esm_master` supports operations on the coupled setup awicm in the versions 1.0, CMIP6 and 2.0; and what the tool can do with that setup. You execute `esm_master` by calling:

```
$ esm_master operation-software-version,
```

e.g.:

```
$ esm_master install-awicm-2.0
```

By default, `esm_master` supports the following operations:

get:
Cloning the software from a repository, currently supporting git and svn

conf:
Configure the software (only needed by mpiesm and icon at the moment)

comp:
Compile the software. If the software includes libraries, these are compiled first. After compiling the binaries can be found in the subfolders `bin` and `lib`.

clean:
Remove all the compiled object files.

install:
Shortcut to get, then conf, then comp.

recomp:
Shortcut to conf, then clean, then comp.

update:
Get the newest commit of the software from the repository.

status:
Get the state of the local database of the software (e.g. `git status`)

log:
Get a list of the last commits of the local database of the software (e.g. `git log`)

To download, compile, and install `awicm-2.0`; you can say:

```
$ esm_master install-awicm-2.0
```

This will trigger a download, if needed a configuration, and a compilation process. Similarly, you can recompile with `recomp-XXX`, clean with `clean-XXX`, or do individual steps, e.g. `get`, `configure`, `comp`.

The download and installation will always occur in the **current working directory**.

You can get further help with:

```
$ esm_master --help
```

11.2 Logging

`esm_master` uses Loguru for console logging. To increase verbosity call `esm_master` with the `-v` or `--verbose` flag.

11.3 Configuring esm-master for Compile-Time Overrides

It is possible that some models have special compile-time settings that need to be included, overriding the machine defaults. Rather than placing these changes in `configs/machines/NAME.yaml`, they can be instead placed in the `computer` section of the component or model configurations using a `choose_general.execution_mode` block, e.g.:

```
computer:
    choose_general.execution_mode:
        compile:
            export_vars:
                SPECIAL_COMPILE_VAR: "compile_value"
```

See *ESM Environment* for more details on environment configuration options.

CHAPTER

TWELVE

ESM-VERSIONS (DEPRECATED)

Note: Deprecated since version 6.0.0: This feature is deprecated since 2022 (version 6.0.0). If you are using a more recent version of ESM-Tools, please use `esm_tools --version` instead, to get the version of *ESM-Tools*. See also section *ESM Tools* and *Usage: Getting the Version*. Please also find further information for e.g. upgrading ESM-Tools here ....

Above version 3.1.5 and below 6.0.0, you will find an executable in your path called `esm_version`. This was added by Paul Gierz to help the user / developer to keep track of / upgrade the python packages belonging to ESM Tools.

12.1 Usage

It doesn't matter from which folder you call `esm_versions`. You have two subcommands:

```
esm_versions check                      gives you the version number of each
                                        installed esm python package
esm_versions upgrade                    upgrades all installed esm python
                                        packages to the newest version
                                        of the release branch
```

Notice that you can also upgrade single python packages, e.g.:

```
esm_versions upgrade esm_parser         upgrades only the package esm_parser
                                        to the newest version of the release
                                        branch
```

And yes, `esm_versions` can upgrade itself.

CHAPTER

THIRTEEN

ESM RUNSCRIPTS

13.1 Usage

```
esm_runscripts [-h] [-d] [-v] [-e EXPID] [-c] [-P] [-j LAST_JOBTYPE]
                  [-t TASK] [-p PID] [-x EXCLUDE] [-o ONLY]
                  [-r RESUME_FROM] [-U] [-i INSPECT]
                  runscript
```

13.2 Arguments

Optional arguments	Description
`-h`, `--help`	Show this help message and exit.
`-d`, `--debug`	Print lots of debugging statements.
`-v`, `--verbose`	Be verbose.
`-e EXPID`, `--expid EXPID`	The experiment ID to use. Default `test`.
`-c`, `--check`	Run in check mode (don't submit job to supercomputer).
`-P`, `--profile`	Write profiling information (esm-tools).
`-j LAST_JOBTYPE`, `--last_jobtype LAST_JOBTYPE`	Write the jobtype this run was called from (esm-tools internal).
`-t TASK`, `--task TASK`	The task to run. Choose from: `compute`, `post`, `couple`, `tidy`.
`-p PID`, `--pid PID`	The PID of the task to observe.
`-x EXCLUDE`, `--exclude EXCLUDE`	E[x]clude this step.
`-o ONLY`, `--only ONLY`	[o]nly do this step.
`-r RESUME_FROM`, `--resume-from RESUME_FROM`	[r]esume from the specified run/step (i.e. to resume a second run you'll need to use `-r 2`).
`-U`, `--update`	[U]pdate the runscript in the experiment folder and associated files
`--update-filetypes UPDATE_FILETYPES [UPDATE_FILETYPES ...]`	Updates the requested files from external sources in a currently ongoing simulation. For example, if you want to update the binaries and the configs (namelists) in a resubmission of a experiment you can do this by adding `--update-filetypes bin config` to your `esm_runscripts` command. We strongly advise against using this option unless you really know what you are doing.
`-i`, `--inspect`	This option can be used to [i]nspect the results of a previous run, for example one prepared with `-c`. This argument needs an additional keyword. Choose among: `overview` (gives you the same litte message you see at the beginning of each run, `lastlog` (displays the last log file), `explog` (the overall experiment logfile), `datefile` (the overall experiment logfile), `config` (the Python dict that contains all information), `size` (the size of the experiment folder), a filename or a directory name output the content of the file /directory if found in the last `run_` folder.)
`--trace`	Enable `TRACE`-level output to stdout.
`--task-log-files`	Enable per-task log files on disk.

13.3 Running a Model/Setup

ESM-Runscripts is the *ESM-Tools* package that allows the user to run the experiments. *ESM-Runscripts* reads the runscript (either a *bash* or *yaml* file), applies the required changes to the namelists and configuration files, submits the runs of the experiment to the compute nodes, and handles and organizes restart, output and log files. The command to run a runscript is:

```
$ esm_runscripts <runscript.yaml/.run> -e <experiment_ID>
```

The `runscript.yaml/.run` should contain all the information regarding the experiment paths, and particular configurations of the experiment (see the yaml:Runscripts section for more information about the syntax of *yaml* runscripts). The `experiment_ID` is used to identify the experiment in the scheduler and to name the experiment's directory (see *Experiment Directory Structure*). Omitting the argument `-e <experiment_ID>` will create an experiment with the default experimant ID `test`.

ESM-Runscript allows to run an experiment check by adding the `-c` flag to the previous command. This check performs all the system operations related to the experiment that would take place on a normal run (creates the experiment directory and subdirectories, copies the binaries and the necessary restart/forcing files, edits the namelists, ...) but stops before submitting the run to the compute nodes. We strongly recommend **running first a check before submitting an experiment to the compute nodes**, as the check outputs contains already valuable information to understand whether the experiment will work correctly or not (we strongly encourage users to pay particular attention to the *Namelists* and the *Missing files* sections of the check's output).

13.4 Job Phases

The following table summarizes the job phases of *ESM-Runscripts* and gives a brief description. ...

13.5 Running only part of a job

It's possible to run only part of a job. This is particularly interesting for development work; when you might only want to test a specific phase without having to run a whole simulation.

As an example; let's say you only want to run the `tidy` phase of a particular job; which will move things from the particular run folder to the overall experiment tree. In this example; the experiment will be called `test001`:

```
esm_runscripts ${PATH_TO_USER_CONFIG} -t tidy
```

13.6 Experiment Directory Structure

All the files related to a given experiment are saved in the *Experiment Directory*. This includes among others model binaries, libraries, namelists, configuration files, outputs, restarts, etc. The idea behind this approach is that all the necessary files for running an experiment are contained in this folder (the user can always control through the runscript or configuration files whether the large forcing and mesh files also go into this folder), so that the experiment can be reproduced again, for example, even if there were changes into one of the model's binaries or in the original runscript.

The path of the *Experiment Directory* is composed by the `general.base_dir` path specified in the runscript (see yaml:Runscripts syntax) followed by the given `experiment_ID` during the `esm_runscripts` call:

```
<general.base_dir>/<experiment_ID>
```

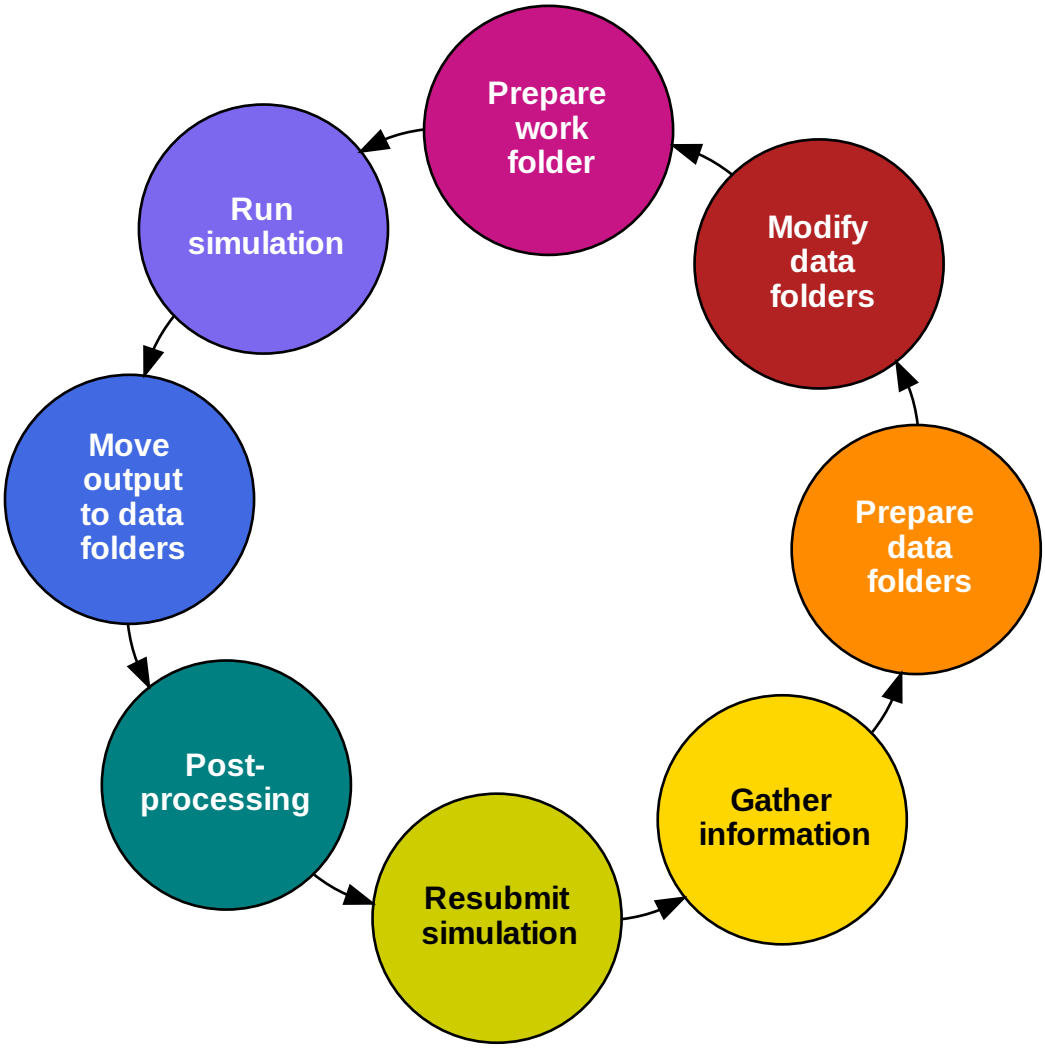


Fig. 1: ESM-Tools job phases

The **main experiment folder** (`General exp dir`) contains the subfolders indicated in the graph and table below. Each of these subfolders contains a folder for each component in the experiment (i.e. for an AWI-CM experiment the `outdata` folder will contain the subfolders `echam`, `fesom`, `hdmodel`, `jsbach`, `oasis3mct`).

The structure of the **run folder** `run_YYYYMMDD-YYYYMMDD` (`Run dir` in the graph) replicates that of the general experiment folder. *Run* directories are created before each new run and they are useful to debug and restart experiments that have crashed.

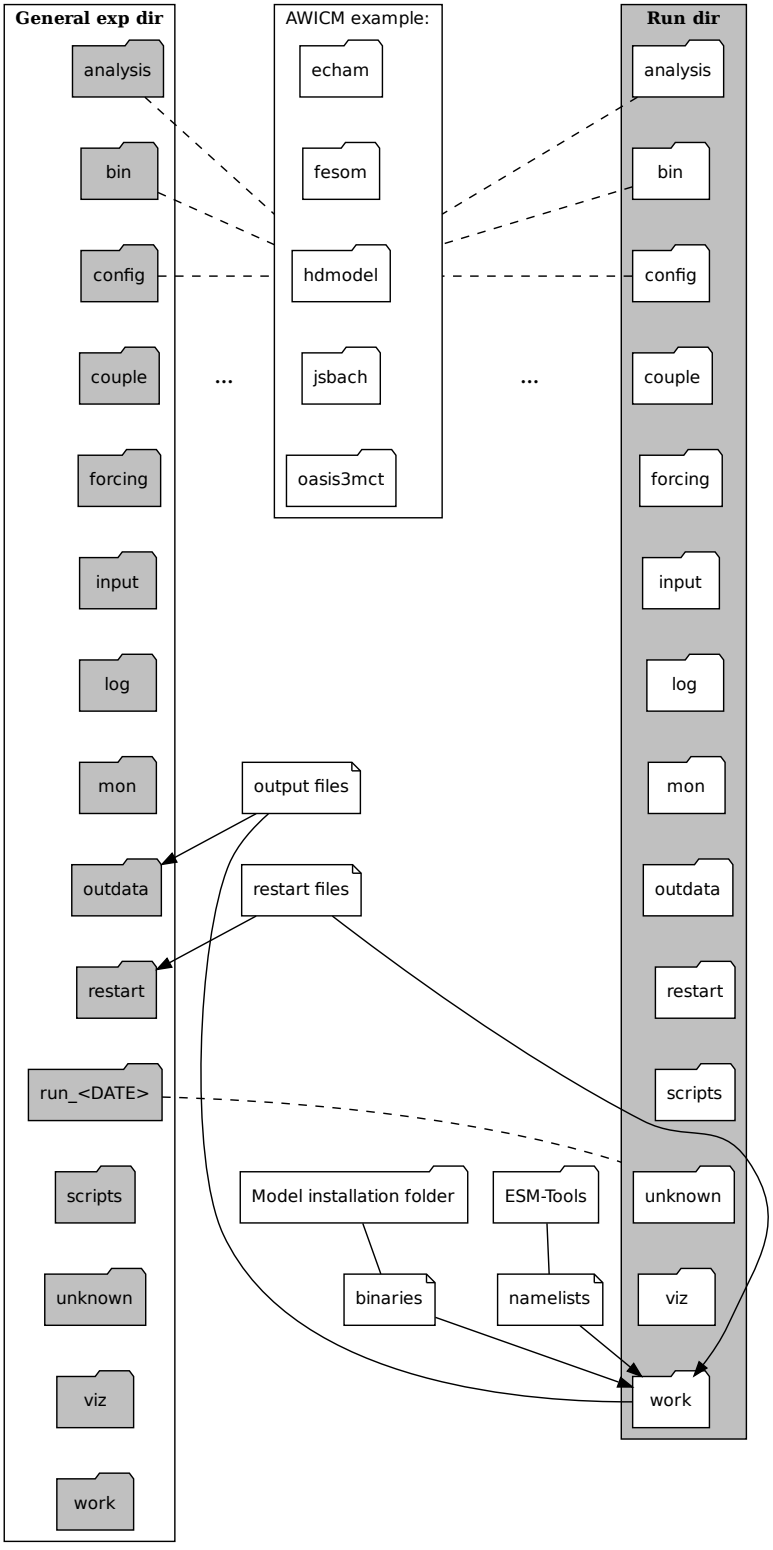


Fig. 2: Experiment directory structure

Subfolder	Files	Description
analysis	user's files	Results of user's "by-hand" analysis can be placed here.
bin	component binaries	Model binaries needed for the experiment.
config	• <experiment_ID>_finished_config.yaml • namelists • other configuration files	Configuration files for the experiment including namelists and other files specified in the component's configuration files (`<PATH>/esm_tools/configs/<component>/<component>.yaml`, see *File Dictionaries*). The file `<experiment_ID>_finished_config.yaml` is located at the base of the `config` folder and contains the whole ESM-Tools variable space for the experiment, resulting from combining the variables of the runscript, setup and component configuration files, and the machine environment file.
couple	coupling related files	Necessary files for model couplings.
forcing	forcing files	Forcing files for the experiment. Only copied here when specified by the user in the runscript or in the configuration files (*File Dictionaries*).
input	input files	Input files for the experiment. Only copied here when specified by the user in the runscript or in the configuration files (*File Dictionaries*).
log	• <experiment_ID>_<setup_name>.log • component log files	Experiment log files. The component specific log files are placed in their respective subfolder. The general log file `<experiment_ID>_<setup_name>.log` reports on the *ESM-Runscripts Job Phases* and is located at the base of the `log` folder. Log file names and copying instructions should be included in the configuration files of components (*File Dictionaries*).
mon	user's files	Monitoring scripts created by the user can be placed here.
outdata	outdata files	Outdata files are placed here. Outdata file names and copying instructions should be included in the configuration files of components (*File Dictionaries*).
restart	restart files	Restart files are placed here. Restart file names and copying instructions should be included in the configuration files of components (*File Dictionaries*).
run_YYYYMMDD	run files	Run folder containing all the files for a given run. Folders contained here have the same names as the ones contained in the general ex-

If one file was to be copied in a directory containing a file with the same name, both files get renamed by the addition of their start date and end dates at the end of their names (i.e. `fesom.clock_YYYYMMDD-YYYYMMDD`).

Note: Having a *general* and several *run* subfolders means that files are duplicated and, when models consist of several runs, the *general* directory can end up looking very untidy. *Run* folders were created with the idea that they will be deleted once all files have been transferred to their respective folders in the *general* experiment directory. The default is not to delete this folders as they can be useful for debugging or restarting a crashed simulation, but the user can choose to delete them (see *Cleanup of run_ directories*).

13.7 Cleanup of `run_` directories

This plugin allows you to clean up the `run_${DATE}` folders. To do that you can use the following variables under the `general` section of your runscript (documentation follows order of code as it is executed):

- `clean_runs`: **This is the most important variable for most users**. It can take the following values:
  - `True`: removes the `run_` directory after each run (**overrides every other** `clean_` **option**).
  - `False`: does not remove any `run_` directory (default) if no `clean_` variable is defined.
  - <int>: giving an integer as a value results in deleting the `run_` folders except for the last <int> runs (recommended option as it allows for debugging of crashed simulations).

  Note: `clean_runs: (bool)` is incompatible with `clean_this_rundir` and `clean_runs: (int)` is incompatible with `clean_old_rundirs_except` (an error will be raised after the end of the first simulation). The functionality of `clean_runs` variable **alone will suffice most of the standard user requirements**. If finer tunning for the removal of `run_` directories is required you can used the following variables instead of `clean_runs`.

- `clean_this_rundir`: (bool) Removes the entire run directory (equivalent to `clean_runs: (bool)`). `clean_this_rundir: True` **overrides every other** `clean_` **option**.
- `clean_old_rundirs_except`: (int) Removes the entire run directory except for the last <x> runs (equivalent to `clean_runs: (int)`).
- `clean_old_rundirs_keep_every`: (int) Removes the entire run directory except every <x>th run. Compatible with `clean_old_rundirs_except` or `clean_runs: (int)`.
- `clean_<filetype>_dir`: (bool) Erases the run directory for a specific filetype. Compatible with all the other options.
- `clean_size`: (int or float) Erases all files with size greater than `clean_size`, must be specified in bytes! Compatible with all the other options.

Example

To delete all the `run_` directories in your experiment include this into your runscript:

```
general:
        clean_runs: True
```

To keep the last 2 `run_` directories:

```
general:
        clean_runs: 2
```

To keep the last 2 runs and every 5 runs:

```
general:
        clean_old_rundirs_except: 2
        clean_old_rundirs_keep_every: 5
```

13.8 Debugging an Experiment

To debug an experiment we recommend checking the following files that you will find, either in the *general* experiment directory or in the *run* subdirectory:

- The *ESM-Tools* variable space file `config/<experiment_ID>_finished_config.yaml`.
- The run log file `run_YYYYMMDD-YYYYMMDD/<experiment_ID>_compute_YYYYMMDD-YYYYMMDD_<JobID>.log`.

For interactive debugging, you may also add the following to the `general` section of your configuration file. This will enable the pdb Python debugger, and allow you to step through the recipe.

```
general:
    debug_recipe: True
```

13.9 Configuration Provenance

In addition to the hints summarized in the "Debugging an Experiment" section, you will also find that the `finished_config.yaml` found in your `config` directory contains end-of-line comments detailing where a particular setting came from. You can use this to better track down what is being set and why, but it is **strongly recommended** that the configuration files found in your `esm-tools` source directory should **not** be changed unless you know exactly what you are doing. All of the configuration settings can be overridden from the run configuration, which is the prefered location for user changes. For more information see *How can I know where a parameter is defined?*.

13.10 Setting the file movement method for filetypes in the runscript

By default, `esm_runscripts` copies all files initially into the first `run_`-folder, and from there to `work`. After the run, outputs, logs, restarts etc. are copied from `work` to `run_`, and then moved from there to the overall experiment folder. We chose that as the default setting as it is the safest option, leaving the user with everything belonging to the experiment in one folder. It is also the most disk space consuming, and it makes sense to link some files into the experiment rather than copy them.

As an example, to configure `esm_runscripts` for an echam-experiment to link the forcing and inputs, one can add the following to the runscript yaml file:

```
echam:
        file_movements:
                forcing:
                        all_directions: "link"
                input:
                        init_to_exp: "link"
                        exp_to_run: "link"
```

(continues on next page)

(continued from previous page)

```
                        run_to_work: "link"
                        work_to_run: "link"
```

Both ways to set the entries are doing the same thing. It is possible, as in the `input` case, to set the file movement method independently for each of the directions; the setting `all_directions` is just a shortcut if the method is identical for all of them.

13.11 Parallel File Movements

By default, `esm_runscripts` moves files (inputs, restarts, outputs, etc.) between experiment directories in parallel using Dask workers distributed across the compute nodes. This can significantly speed up the file-handling phases of a simulation, especially on large allocations.

The mode is controlled by `parallel_file_movements` in the `general` section of your runscript. To use Dask workers across compute nodes (the default) use:

```
general:
    parallel_file_movements: "dask"
```

To use local threads instead (only 1 node with cores working in parallel):

```
general:
    parallel_file_movements: "threads"
```

To disable parallel file movements entirely (serial):

```
general:
    parallel_file_movements: False
```

13.11.1 Which mode should I use?

- **Small runs (< 5 nodes)**: `"threads"` is usually sufficient and has no cluster initialization overhead.
- **Large runs (>= 5 nodes)**: `"dask"` distributes I/O across all nodes and scales better. The small startup cost is offset by faster file transfers.
- **Debugging or safety**: `False` runs everything serially.

Note: When `"dask"` is selected but the Dask cluster is not available (e.g. on a login node or if workers fail to start), `esm_runscripts` automatically falls back to `"threads"` with a warning.

13.11.2 Dask internals

When `parallel_file_movements` is set to `"dask"`, the following happens automatically:

1. **Cluster startup** – Before the main recipe begins, a Dask scheduler and workers are launched via `srun` (SLURM) on the allocated nodes. The scheduler writes a `dask_scheduler.json` file into the run's `work` directory.
2. **Parallel I/O** – During each file-movement phase, `esm_runscripts` connects to the Dask cluster and submits copy/link/move operations as parallel tasks. If any single file transfer fails, it is retried serially as a fallback.

The cluster is configured through the `dask` section, which has sensible defaults but can be tuned in your runscript:

Table 1: Dask configuration variables

Variable	Default	Description
`dask.client_timeout`	`0.05`	Timeout (seconds) when probing the Dask scheduler status.
`dask.workers_timeout`	`5`	Max time (seconds) to wait for workers to become available.
`dask.poll_interval`	`0.5`	How often (seconds) to poll for cluster readiness.
`dask.init_scheduler_cmd`	per batch system	Shell command to start the Dask scheduler (defined in the batch system YAML, e.g. `slurm.yaml`).
`dask.init_workers_cmd`	per batch system	Shell command to start the Dask workers (defined in the batch system YAML, e.g. `slurm.yaml`).
`dask.scheduler_json`	`${general.thisrun_work_dir}/dask_scheduler.json`	Full path to the Dask scheduler JSON file used for client connections.
`dask.actions`	`["parallel_file_movements"]`	List of actions that trigger Dask cluster initialization.

The Dask scheduler and worker launch commands are defined per batch system (e.g. in `slurm.yaml`) and are not typically changed by users. For SLURM, the default worker count is `nnodes * partition_cpn / 4`, meaning one Dask worker per four CPU cores (see `configs/other_software/batch/slurm.yaml`). Workers are distributed cyclically across all allocated nodes using InfiniBand:

slurm.yaml

```
dask:
    ntasks: "$(( ${computer.nnodes} * ${computer.partition_cpn} / 4 ))"
    ...
    init_workers_cmd: "srun --ntasks=${dask.ntasks} --cpus-per-task=1 --nodes=@nodes@ --
↪distribution=cyclic:cyclic:cyclic dask worker --scheduler-file ${dask.scheduler_json} -
↪-nthreads 1 --nworkers 1 --interface ib0"
```

If you need to change the number of workers, you can either redefine `dask.ntasks` or provide a custom `dask.init_workers_cmd` in any of your configuration files or directly in your runscript.

13.12 Running an experiment with a virtual environment

Running jobs can optionally be encapsulated into a virtual environment.

To use a virtual environment run `esm_runscripts` with the flag `--contained-run` or set `use_venv` within the `general` section of your runscript to `True`:

```
general:
    use_venv: True
```

This shields the run from changes made to the remainder of the ESM-Tool installation, and it's strongly recommended for production runs.

> **Warning:** Refrain from using this feature if you have installed ESM-Tools within a conda environment.

If you choose to use a virtual environment, a local installation will be created in the experiment tree at the begining of the first run into the folder named `.venv_esmtools`. **That** installation will be used for the experiment. It will be installed at the root of your experiment and contains all the Python libraries used by ESM-Tools. The installation at the beginning of the experiment will induce a small overhead (~2-3 minutes).

For example, for a user `miguel` with a run with *expid* `test` ESM-Tools will be installed here:

```
/scratch/miguel/test/.venv_esmtools/lib/python3.10/site-packages/esm_tools
```

instead of:

```
/albedo/home/miguel/.local/lib/site-packages/esm_tools
```

The virtual environment installs by default the `release` branch, pulling it directly from our GitHub repository. You can choose to override this default by specifying another branch, adding to your runscript:

```
general:
    install_esm_tools_branch: '<your_branch_name>'
```

> **Warning:** The branch **needs to exist on GitHub** as it is cloned form there, and **not from your local folder**. If you made any changes in your local branch make sure they are pushed before running `esm_runscripts` with a virtual environment, so that your changes are included in the virtual environment installation.

You may also select to install esm_tools in *editable mode*, in which case they will be installed in a folder `src/esm_tools/` in the root of your experiment. Any changes made to the code in that folder **will** influence how ESM-Tools behave. To create a virtual environment with ESM-Tools installed in *editable* mode use:

```
general:
    install_<esm_package>_editable: true/false
```

Note: When using a virtual environment, config files and namelists will come of the folder .venv_esmtools listed above and **not** from your user install directory. You should make **all** changes to the namelists and config files via your user runscript (*Changing Namelists*). This is recommended in all cases!!!

13.13 Running an experiment with conda

If you submit `esm_runscripts` from within an active conda environment, that same environment is automatically activated inside the generated job script before the model runs. **You don't need to do anything** for this to work.

If you need finer control (e.g. the environment used on the compute nodes should differ from the one used to launch `esm_runscripts`, or conda is not on the default `PATH` of the compute nodes), you can specify it explicitly in a `conda` section of your runscript:

```
conda:
    env: /path/to/your/conda/env
    root: /path/to/conda  # optional, needed if "conda" is not already on PATH
```

See `conda.env`, `conda.root`, and `launched_with_conda` in *Run-time variables* for details.

Warning: Do not combine this feature with the virtual environment feature described above (`use_venv`); mixing a conda environment with a venv created by ESM-Tools may cause conflicts.

13.14 Logging and verbosity

`esm_runscripts` uses Loguru-based logging with simple flags to control verbosity and file logging. Logs are always written in the main run log (`<base_dir>/<expid>/log/<expid>_<model>_<datestamp>_<jobid>.log`). For more log granularity, it is possible to also set `--task-log-files` as a flag of `esm_runscripts`, to write logs of each task to a separate file. You can use the following `esm_runscripts` flags to control the logging behavior:

- `--trace`: enable `TRACE`-level output to stdout. Prints very detailed diagnostics and the parsed command-line config.
- `-d`, `--debug`: enable `DEBUG`-level output to stdout (less detailed than `--trace`) and breakpoints.
- `-v`, `--verbose`: also enables `DEBUG`-level output to stdout, without breakpoints.
- `--task-log-files`: enable per-task log files on disk. When enabled, `esm_runscripts` writes each task's output to a file in the experiment's `log` folder (`<base_dir>/<expid>/log/<expid>_<model>_<task>_<datestamp>_<jobid>.log`). To reduce the number of files, this option is turned off by default, but the logs are always printed in the run log anyway.

Note: Because the logging starts before the parsing of the yaml files, it is not possible to control the logging behavior from variables defined in the yamls. Only command-line flags can control the logging behavior.

CHAPTER

FOURTEEN

ESM RUNSCRIPTS - USING THE WORKFLOW MANAGER

14.1 Introduction

Starting with Release 6.0, `esm_runscripts` allows to define additional *jobs* for e.g. data processing, coupling. Such jobs can be arranged into job-clusters, and the order of execution can be set in a flexible and short way from the runscript. This is applicable for both pre- and postprocessing, but especially useful for iterative coupling jobs, like e.g. coupling PISM to VILMA (see below). In this section we explain the basic concept, describe the keywords that have to be set in the runscript in order to make use of this feature, and give some examples on how to integrate pre- and postprocessing jobs and how to set up jobs for iterative coupling.

14.2 Default jobs of a general model simulation run

The task of `esm_runscript` is split into different subjobs which are:

```
newrun --> prepcompute --> compute --> observe_compute --> tidy (+ resubmit next run)
```

These standard jobs are all separated and independent, each submitted (or started) by the previous job in one of three ways (see below). Here is what each of the standard jobs do:

Job	Description	Started by
newrun	Initializes a new experiment, only very basic stuff, like creating (empty) folders needed by any of the following subjobs/jobs. **Warning:** It needs to be the first job of any *experiment*.	
prepcompute	Prepares the compute job. All the (Python) functionality that needs to be run, up to the job submission. Includes copying files, editing namelists, write batch scripts, etc.	newrun
compute	Actual model integration, nothing else. No Python codes involved.	prepcompute via `sbatch` or other batch system command
observe_compute	Python job running at the same time as compute, checking if the compute job is still running, looking for some known errors for monitoring / job termination.	`sbatch`, started by its own `esm_runscripts` call in the `.run` script, after the `compute` job has been submitted with `srun` or other batch launcher.
tidy	Sorts the produced outputs, restarts and log files into the correct folders, checks for missing and unknown files, builds coupler restart files if not present	observe_compute

Note: None of this has to be edited by the users. The above described workflow jobs form the default set of jobs

needed to run any simulation. Changing anyone of these jobs may lead *ESM-Tools* to fail. However, additional jobs can be added to this workflow, as described below, to extend the default workflow.

14.3 Inspect workflow jobs

To inspect the workflow and workflow jobs that are defined by e.g. a choosen setup or in an already run simulation/experiment, you can run esm_runscript with the `-i` (`--inspect`) option. This can be done for two different cases:

- To inspect the workflow previous to running a certain experiment. For example, if you want to add a new workflow job, and need to know which jobs are already defined in a choosen setup or model configuration:

```
esm_runscripts runscript.yaml -i workflow
```

- To inspect a workflow from an experiment that has beed carried out already or created during a check-run (-c):

```
esm_runscripts runscript.yaml -e <expid> -i workflow
```

It will display the workflow configuration showing the order of workflow jobss and their attributes and possible dependencies. This output should help to find out the correct keyworkds to be set when integrating a new workflow job.

Example output:

```
Workflow sequence (cluster [jobs])
----------------------------------
prepcompute ['prepcompute'] ->  compute ['compute'] ->  tidy ['tidy'] ->  prepcompute [
↪'prepcompute'] and my_own_new_cluster ['my_new_last_job', 'my_second_new_job']
```

14.4 Defining additional workflow jobs

If it is necessary to complement the default workflow with simulation specific processing steps, this sequence of default workflow jobs can be extended by adapting the runscipt or any component specific configuration files. The workflow manager will evaluate these additional jobss and integrate them into the default sequence of the workflow. In order to integrate the additional jobs correctly, the following information about this job needs to be given in the one of the yaml files:

- Name of the script to be run
- Name of the python script used for setting up the environment
- Path to the directory in which both of the above scripts can be found
- Information on how often the job should be called
- Information where in the workflow the new job needs to be inserted
- In case it isn't clear, which job should resubmit the next run.

In general, a workflow can be defined in the runscript or in any component configuration file. But there are some restrictions to the definition that needs to be taken into account:

- The name of each job needs to be unique. Otherwise, an exception error will be raised.
- The names of the default jobs are not allowed to be used for any new jobs. This will also cause an exception error during runtime.

- Settings in the runscript will overwrite settings in other config files. (See also *Hierarchy of YAML configuration files*.)

14.4.1 Keywords to define a new workflow job

To provide the information about a new job the following keywords and mappings (key/value pairs) are available (keywords that are indicated with < > need to be adapted by the user):

Keyword	Mandatory	(Default) values	Function
`workflow`	yes	–	Chapter headline in a runscript or configuration section, indicating that alterations to the standard workflow will be defined here.
`subjobs`	yes	user defined string	Section within the `workflow` chapter that containes new additional workflow jobs.
`<new_job_name>`	yes	user defined string	Section within the `subjobs` section for each new job. The name of the new job needs to be unique. See also further explenation in *Defining additional workflow jobs*
`run_after: <value>` or `run_before: <value>`	no	default: last job in (default) workflow (e.g. tidy)	Key/value entry in each `job` section. This mapping defines the (default or user) job of the workflow after or before the new job should be executed. Only one of the two should be specified.
`submit_to_batch_system: <value>`	no	**false**, true	Key/value entry in each `job` section. This mapping defines if the (default or user) job is submitted to batch system or not.
`run_on_queue: <value>`	no	None	Key/value entry in each `job` section. This mapping defines to which queue (name) the job should be submitted to.
`batch_or_shell: <value>`	no	**shell**, batch	Key/value entry in each `job` section. This mapping defines if the (default or user) job is submitted as batch job or as shell script. This attribute will be overwritten depending on `submit_to_batch_system`
`cluster: <value>`	no	Job name	Key/value entry in each `job` section. Jobs that have the same entry in `cluster` will be run from the same batch script.
`order_in_cluster: <value>`	no	**sequential**, concurrent	Key/value entry in each `job` section. This mapping defines how jobs in the same `<cluster>` should be run. Concurrent or serial.
`script: <value>`	yes	None	Key/value entry in each `job` section. This mapping defines the name of the script that is going to be executed during the new workflow job.
`script_dir: <value>`	yes	None	Key/value entry in each `job` section. This mapping defines the path to the script set by the variable `<script>`.
`call_function: <value>`	no	None	Key/value entry in each `job` section. This mapping defines the function within the script defined in variable `<script>` should be executed.
`env_preparation: <value>`	no	None	Key/value entry in each `job` section. This mapping defines e.g. a Python script/function that prepares a dictionary with environment variables.
`nproc: <int>`	no	1	Key/value entry in each `job` section. This mapping defines the number of CPUs a job should run with (if run via sbatch).
`run_only: <value>`	no	None	Key/value entry in each `job` section. This mapping defines when the job should be run. E.g. run only at the beginning of a *chunk* (set of runs).
`skip_chunk_number: <int>`	no	None	Key/value entry in each `job` section. This mapping defines how many chunks should be skipped before the job will be execited.
`trigger_next_run: <value>`	no	**false**, true	If job should trigger next run

14.4.2 Syntax example

The following code snippet shows the general syntax for defining a new workflow job:

```
workflow:
    subjobs:
        <job_name>:
            run_after: <value>
            submit_to_batch_system: <value>
            run_on_queue: <value>
            cluster: <value>
            order_in_cluster: <value>
            script: <value>
            call_function: <value>
            env_preparation: <value>
            nproc: <value>
            run_only: <value>
            skip_chunk_number: <value>
            trigger_next_run: <value>
```

14.5 Workflow defaults

A minimal example of defining a new workflow job is given in Example 1. This will integrate a new job with the following default assumptions:

- The new job will be run after the last job of the default workflow.
- The script given for this job is run as a subprocess (not a batch run).
- The next run of the overall experiment will be (still) triggered by the last job of the default workflow and not the new job.

14.6 Examples for the definition of new workflow jobs

14.6.1 Example 1: Adding an additional postprocessing subjob

In the case of a simple postprocessing task (here for model Echam), that sould be run as the last task of each run, independantly from restarting the experiment, the corresponding minimal code snippet in a runscript could look like this:

```
echam:
    [...other information...]

    workflow:
        subjobs:
            my_postprocessing:
                script_dir: <value>
                script: <values>
```

14.6.2 Example 2: Adding an additional preprocessing subjob

A preprocessing job basically is configured the same way as a postprocessing job, but the run_before keyword is needed now, to define when the new job should be run:

```
echam:
    [...other information...]

    workflow:
        subjobs:
            my_preprocessing:
                run_before: prepcompute
                script_dir: <value>
                script: <values>
```

14.6.3 Example 3: Adding a new job as the last task in a run

To integrate a new job that should be run as the last task in every run but before the next run starts, use the following example:

```
echam:
    [...other information...]

    workflow:
        subjobs:
            my_new_last_job:
                script_dir: <value>
                script: <values>
                trigger_next_run: True
```

14.6.4 Example 4: Adding multiple user jobs that can be run concurrently in a workflow cluster

It is possible to define multiple new jobs that should start at the same but can be run independently from each other. This can be done by assigning these jobs to the same workflow cluster and run them concurrently over the batch system:

```
echam:
    [...other information...]

    workflow:
        subjobs:
            my_new_last_job:
                script_dir: <value>
                script: <values>
                submit_to_batch_system: True
                run_on_queue: <value>
                cluster: my_own_new_cluster

            my_second_new_job:
                script_dir: <value>
                script: <values>
```

(continues on next page)

(continued from previous page)

```
                submit_to_batch_system: True
                run_on_queue: <value>
                cluster: my_own_new_cluster
```

14.6.5 Example 5: Adding an iterative coupling job

Writing a runscript for iterative coupling using the workflow manager requires some more changes. The principal idea is that each coupling step consists of two data processing jobs, one pre- and one postprocessing job. This is done this way as to make the coupling modular, and enable the modeller to easily replace one of the coupled components by a different implementation. This is of course up to the user to decide, but we generally advise to do so, and the iterative couplings distributed with *ESM-Tools* are organized this way. :

```
echam:
    [...other information...]

     workflow:
        subjobs:
            couple_in:
                nproc: 1
                run_before: prepcompute
                script: coupling_ice2echam.functions
                script_dir: ${general.script_dir}/echam
                call_function: ice2echam
                env_preparation: env_echam.py
                run_only: first_run_in_chunk
                skip_chunk_number: 1

            couple_out:
                nproc: 1
                run_after: tidy
                script: coupling_echam2ice.functions
                script_dir: ${general.script_dir}/echam
                call_function: echam2ice
                env_preparation: env_echam.py
                run_only: last_run_in_chunk
                trigger_next_run: True

fesom:
    [...other information...]

    workflow:
        subjobs:
            couple_in:
                nproc: 1
                run_before: prepcompute
                script: coupling_ice2fesom.functions
                script_dir: ${general.script_dir}/fesom
                call_function: ice2fesom
                env_preparation: env_fesom.py
                run_only: first_run_in_chunk
                skip_chunk_number: 1
```

(continues on next page)

(continued from previous page)

```
        couple_out:
            nproc: 1
            run_after: tidy
            script: coupling_fesom2ice.functions
            script_dir: ${general.script_dir}/fesom
            call_function: fesom2ice
            env_preparation: env_fesom.py
            run_only: last_run_in_chunk
            trigger_next_run: True
```

CHAPTER

FIFTEEN

ESM ENVIRONMENT

The package `esm_environment` takes care of generating the environments for the different HPCs supported by *ESM-Tools*. This is done through the use of the `EnvironmentInfos` class inside the different *ESM-Tools* packages.

For the correct definition of an HPC environment a *yaml* file for that system needs to be included inside the `esm_tools` package inside the `configs/machines/` folder (e.g. `levante.yaml`). This file should contain all the required preset variables for that system, and optionally, the environment variables `general_actions`, `module_actions`, `spack_actions`, `export_vars`, and `unset_vars`. These environment variables are defined in the `computer` section/key of the configuration.

Hint: To get started easily, you can use the `esm_tools` command to create machine files, component files, or setup files, which include examples of the environment variables in the `computer` section:

```
# Create a new machine configuration
esm_tools create-new-config -t machine NAME

# Create a new component configuration
esm_tools create-new-config -t component NAME

# Create a new setup configuration
esm_tools create-new-config -t setup NAME
```

15.1 Environment variables

Environment variables must be defined inside the `computer` section of your YAML configuration files. These can be *machine* files, *component* files, *setup* files, or *runscripts*. All environment-related keys should be placed inside this section.

```
computer:
    # Environment variables go here
    module_actions: [...]
    export_vars: {...}
```

The following environment variables are supported:

general_actions (list)
: A list of general actions to be included in the compilation and run scripts. These are added directly to the script without any prefix.

```
computer:
    general_actions:
        - "echo 'Starting environment setup'"
```

module_actions (list)
A list of `module` actions to be included in the compilation and run scripts generated by `esm_master` and `esm_runscripts` respectively, such as `module load netcdf`, `module unload netcdf`, `module purge`, etc. The syntax of this list is such as that of the command that would be normally used in shell, but omitting the `module` word, for example:

```
computer:
    module_actions:
        - "purge"
        - "load netcdf"
```

This variable also allows for sourcing files by adding a member to the list such as `source <file_to_be_sourced>`. You can combine module loads with sourcing files in the same list, as shown in the following example:

```
computer:
    module_actions:
        - "source /path/to/module/conf.sh"
        - "load netcdf/4.7.4"
```

spack_actions (list)
A list of `spack` actions to be included in the compilation and run scripts. Similar to `module_actions`, but for *Spack* package manager commands.

```
computer:
    spack_actions:
        - "load python"
        - "load netcdf-c"
```

export_vars (dict)
A dictionary containing all the variables (and their values) to be exported. The syntax is as follows:

```
computer:
    export_vars:
        A_VAR_TO_BE_EXPORTED: the_value
```

The previous example will result in the following export in the script produced by `esm_master` or `esm_runscripts`:

```
export A_VAR_TO_BE_EXPORTED=the_value
```

As a dictionary, `export_vars` is not allowed to have repeated keys. This could be a problem when environments are required to redefine a variable at different points of the script. To overcome this limitation, repetitions of the same variable are allowed if the key is followed by an integer contained inside `[(int)]`:

```
computer:
    export_vars:
        A_VAR_TO_BE_EXPORTED: the_value
        "A_VAR_TO_BE_EXPORTED[(1)]": $A_VAR_TO_BE_EXPORTED:another_value
```

The resulting script will contain the following exports:

```
export A_VAR_TO_BE_EXPORTED=the_value
export A_VAR_TO_BE_EXPORTED=$A_VAR_TO_BE_EXPORTED:another_value
```

Note that the index is removed once the exports are transferred into the script.

unset_vars (list)
A list of variables to be unset in the script.

```
computer:
    unset_vars:
        - "OLD_VAR"
        - "ANOTHER_OLD_VAR"
```

The resulting script will contain:

```
unset OLD_VAR
unset ANOTHER_OLD_VAR
```

15.2 Using choose blocks with `general.execution_mode`

ESM-Tools automatically sets `general.execution_mode` to either `compile` or `run` depending on the current execution mode. You can leverage this to create execution-mode-specific environments by using `choose_` blocks. This common pattern allows you to define different environment variables for compilation and runtime scripts:

```
computer:
    choose_general.execution_mode:
        compile:
            # These environment variables are only included during compilation
            module_actions:
                - "load intel/19.1.3"
                - "load netcdf-fortran/4.5.3"
            export_vars:
                NETCDF_ROOT: "/path/to/netcdf"
                FORTRAN_COMPILER: "ifort"
        run:
            # These environment variables are only included during runtime
            module_actions:
                - "load intel/19.1.3"
                - "load hdf5/1.12.1"
            export_vars:
                OMP_NUM_THREADS: "4"
                IO_MODE: "async"
```

This approach allows you to maintain separate environments for compilation and runtime, which is often necessary as the requirements may differ between these phases.

Note: You can also nest `choose_` blocks for more granular control. For example, you could combine `choose_general.execution_mode` with `choose_computer.name` to have different compilation and runtime environments for different machines.

15.3 Modification of the environment through the model/setup files

As previously mentioned, the default environment for a HPC system is defined inside its *machine* file (in `esm_tools/machines/<machine_name>.yaml`). However, it is possible to modify this environment directly through the `computer` section of the *component* and/or *setup* files (or even inside the runscript) to adjust to the *component*/*setup* requirements.

Note: The variables `environment_changes`, `compiletime_environment_changes`, and `runtime_environment_changes` are now **deprecated**. Instead, define environment variables directly in the `computer` section using the `add_` prefix.

To add to or modify the environment, use the following variables in the `computer` section:

add_general_actions (list)
: Adds actions to the `general_actions` list.

add_module_actions (list)
: Adds actions to the `module_actions` list.

add_spack_actions (list)
: Adds actions to the `spack_actions` list.

add_export_vars (dict)
: Adds variables to the `export_vars` dictionary.

add_unset_vars (list)
: Adds variables to the `unset_vars` list.

The syntax for these environment variables is the same as their counterparts without the `add_` prefix. These variables can be nested inside `choose_` blocks:

```
computer:
        choose_computer.name:
                ollie:
                        add_export_vars:
                                COMPUTER_VAR: 'ollie'
                juwels:
                        add_export_vars:
                                COMPUTER_VAR: 'juwels'
```

Note: These changes are model-specific for compilation by default, meaning that **the changes will only occur for the compilation script of the model containing those changes**. For runtime, all the environments of the components will be added together into the same `.run` script. Please, refer to *Coupled setup environment control* for an explanation on how to control environments for a whole setup.

15.4 Order of environment variables in configuration files

The placement of a `choose_` block in relation to other environment variables (whithin the same file) controls the order in which they appear in the generated script. Variables defined above a `choose_` block will appear before those defined inside the block, while variables defined after the block will appear after. This allows you to control the sequence of environment operations, which can be important when certain variables depend on others.

```
general:
    execution_mode: run
computer:
    name: levante
    # These will be processed FIRST
    choose_computer.name:
        levante:
            add_export_vars:
                VAR_A: 1

    # These will be processed NEXT
    add_export_vars:
        VAR_B: 2

    # These will be processed LAST
    choose_general.execution_mode:
        run:
            add_export_vars:
                VAR_C: 3
```

With this example, the generated script would contain these exports in the following order:

```
export VAR_A=1
export VAR_B=2
export VAR_C=3
```

This is because the refactored environment processing system preserves the order from the original configuration file, while resolving the appropriate choices based on the context.

Note: The ordering described above only applies to environment variables defined within the same file. The order of environment variables across different files (e.g., machine, component, and setup files) is determined by the configuration hierarchy and cannot be directly controlled. Variables from higher priority files (according to the *YAML File Hierarchy*) will appear in the script before variables from lower priority files.

15.5 Coupled setup environment control

Note: The features described in this section are advanced features that most users will not need to use. They are primarily intended for ESM-Tools developers or experienced users who need to manage complex environment configurations in coupled setups. They provide greater flexibility and control over how environments are managed across components.

15.5.1 Removing environment variables from *component* files

When working with coupled setups, you may want to exclude environment variables that are defined in *component* configuration files. There are two ways to do this:

1. **Global setting**: Set `include_env_from_component_files: false` in the main `computer` section of your *setup* configuration to exclude environment variables from all *component* files.
2. **Component-specific setting**: Set `include_env_from_component_files: false` in a specific *component* section of your *setup* configuration to exclude environment variables only from that *component*'s files.

This feature is particularly useful when you want complete control over the environment in a coupled *setup*, overriding any *component*-specific environment settings that might conflict with your *setup* requirements.

Example: Disabling component environment variables

```
# In your coupled setup configuration (e.g., awicm.yaml)
general:
    setup_name: awicm

computer:
    # Define setup-wide environment instead
    add_export_vars:
        SETUP_VAR: "setup_value"

echam:
    # Component-specific setting: Only exclude environment variables from echam␣
↪component file
    include_env_from_component_files: false
    add_export_vars:
        ECHAM_VAR: "echam_value"
```

In this example:

1. The *setup* file defines general environment variables in the main `computer` section that apply to all *components*
2. For the `echam` *component* specifically, we set `include_env_from_component_files: false` to exclude any environment variables defined in the `echam` *component* file
3. The environment variables defined directly in the `echam` section of the *setup* file (`ECHAM_VAR`) are still included

15.5.2 Environment merging behavior

The merging behavior for environments in coupled setups is controlled by the `merge_component_envs` setting in the `computer` section:

```
computer:
    merge_component_envs:
        compile: false  # Component-specific for compilation
        run: true       # Merged for runtime (default)
```

When set to `true` (default for runtime), environments from all components are merged into a single environment. When set to `false` (default for compilation), each component maintains its own environment.

15.5.3 Advanced control with attributes

In a setup, you might want to have fine-grained control of the environment variables that you add to the `computer` section, so that you can indicate that they belong to an specific component or that they need to be only added during the compilation or runtime. For this, you can use environment variable attributes to specify which components and execution modes they apply to. For this purpose, the following attributes can be specified inside an environment variable, instead of only its value:

- `_value`: The actual value of the environment variable
- `_execution_mode`: The execution mode for which this variable applies ("compile" or "run")
- `_component`: The component for which this variable applies

If the current execution mode or component doesn't match the specified attributes, the variable will take the value defined in `_old_value` (if it exists) or be excluded altogether.

Example 1: Variable that only applies during FESOM runtime:

```
computer:
    export_vars:
        MODEL_SPECIFIC_VAR:
            _value: "some_value"
            _component: "fesom"     # Only applies to FESOM
            _execution_mode: "run"  # Only during runtime
```

With this configuration, when running FESOM (and only in this case), the variable would appear in the run script as:

```
export MODEL_SPECIFIC_VAR="some_value"
```

In all other cases (compiling FESOM, compiling ECHAM, or running ECHAM), this variable would not appear in the generated scripts because the component and/or execution mode attributes don't match.

Example 2: Variable that only applies during OpenIFS compilation:

```
computer:
    export_vars:
        OIFS_OASIS_BASE:
            _value: /path/to/oasis
            _execution_mode: compile
            _component: oifs
```

This variable would only be included in the compilation script for OpenIFS, and would be excluded in all other cases.

CHAPTER

SIXTEEN

ESM-TESTS

Note: This is a feature aimed for advance users and developers who work in preparing default configurations of experiments, or who implement a model/coupled-setup in ESM-Tools.

ESM-Tests is the integration testing suite from *ESM-Tools*. Its aim is to test a set of selected runscripts and model builds just by using one single command: `esm_tests`. It can also perform dry actions (i.e. check compilations and check runs).

16.1 Glossary

actual vs check test
: An *actual test* is a test where the model has been compiled and run in one of the supported HPCs. A *check test* is a dry test, meaning, no compilation or run takes place, but instead, the configuration files and scripts are generated. Both *actual* and *check* tests compare their output configuration files and scripts to the *last-state*, and offer the possibility to update the *last-state* of those files at the end of the test.

esm_tests_info
: The repository where the files of the `last-state` and the `runscripts` for testing are stored. This repository is clone as a submodel of *ESM-Tools* whenever `esm_tests` command is run for the first time. The repository is cloned locally into the `esm_tools/src/esm_tests/resources` folder. You can activate the submodule manually via `git submodule init` followed by `git submodule sync`.

last-state
: Set of configuration files, both from compilation and runtime, that represent the last approved configuration of the testing runscripts. This set of files is kept for comparison with the equivalent files of future pull-requests, to ensure the stability of the configurations. *ESM-Tests* always compares the new files to the `last-state` files automatically, both in actual compilation/runs or check compilation/runs. See *Last-state*.

runscripts
: The runscripts to run the tests. Runscripts define the test simulation details as in regular *ESM-Tools* runscripts, but are also used for *ESM-Tests* to understand what needs to be compiled. Runscripts are part of the `esm_tests_info` submodule, and can be found (if the submodule was initiated via `esm_tests -u`) in `esm_tools/src/esm_tests/resources/runscripts`. Runscripts need to be generalized (i.e. `choose_computer.name`) for the different HPCs where you want to run the tests.

`state.yaml`
: In this document some times referred only as state, is a *YAML* file that includes information about the status of the tests (actual tests or check tests) in different computers. It also includes the date of the last actual test.

16.2 Usage

ESM-Tests is designed to compile and run tests **just with one single command**, without additional arguments: `esm_tests`, so that launching a suite of tests in a supported HPC is straight forward. Higher granularity in the control of the tests is enabled via:

- *Arguments*
- Runscripts via the usual `esm_parser` syntax (e.g. `choose_computer.name`)
- *Model control file (config.yaml)*
- *Local test configuration (test_config.yaml)*

The commands syntax is as follows:

```
esm_tests [-h] [-n] [-c] [-u] [-d] [-s SAVE] [-t] [-o] [-b] [-g] [-e] [-r BRANCH]
```

16.3 Arguments

Optional arguments	Description
-h, –help	Show this help message and exit
-n, –no-user	Avoid loading user config (for check tests in GitHub)
-c, –check	Check mode on (does not compile or run, but produces some files that can be compared to previous existing files in `last_tested` folder)
-u, –update	Updates the resources with the release branch, including runscriptsand last_tested files
-d, –delete	Delete previous tests
-s SAVE, –save SAVE	Save files for comparisson in `last_tested` folder. The values can be `True/False`
-t, –state	Print the state stored in `state.yaml`
-o, –hold	Hold before operation, to give time to check the output
-b, –bulletpoints	Bullet points for printing the state and copy/paste as markdown text
-g, –github	Use this flag when running in GitHub servers (i.e. adds syntax for collapsing compare sections of the output for GitHub Actions)
-e, –system-exit-on-errors	Trigger a system exit on errors or file differences so that GitHub actions can catch that as a failing test
-r BRANCH, –branch BRANCH	use the given esm_tests_info branch

16.4 Last-state

The `last-state` files are https://github.com/esm-tools/esm_tets_info repository, in the `release` branch. The files stored in the `last-state` are: * compilation scripts (`comp-*.sh`) * namelists * namcouple * finished_config * batch scripts (`.run`)

16.5 Check test status

As a user, you can check the `last-state` status (the online one of the `esm_tests_info` repo, `release` branch) by running:

```
esm_tools test-state
```

This will give you a summary of the state of compilation and running tests for different models, in different computers, and also a date of when the latest actual compilation and run tests were carried out.

If you are testing locally in an HPC, you can get the same information about your local state by running:

```
esm_tests -t
```

16.6 Model control file (`config.yaml`)

File location: `esm_tools/src/esm_tests/resources/runscripts/<model>/config.yaml` **Versioned**: Yes, distributed with `esm_tests_info`

The *Model control file* gives you control over *ESM-Tests* setups for the set of runscripts for a given model (the model which name is the same as the folder where the `config.yaml` is contained: `esm_tools/src/esm_tests/resources/runscripts/<model>/`).

Within this file you can control:

- which files need to be present for considering an `actual compilation test` successful (`comp.actual.files`)
- which files need to be present for considering an `actual run test` successful (`run.actual.files`)
- which messages from the execution of `esm_runscripts` should trigger an error in an `actual run test` (`run.actual.errors`)
- which `computers` are supported for this set of tests (`computers`)

The file should contain this structure:

```
comp:
        actual:
                files:
                    - "file/path" # Typically the binaries
        check: {}
run:
        actual:
                errors:
                    - "error message to mark the test as not successful # Typically
↪"MISSING FILES"
                files: # Typically restart files and outdata files
                    - "path/to/file1"
                    - "globbing/path/*/to*files"
        check: {}
computers:
        - <computer1>
        - <cimputer2>
```

In the `files` sections, **globbing is supported**.

The file's paths should be relative to the compilation folder or the experiment folder.

Each file name can be followed by the syntax `in/except [<model_version1>, <model_version2>, ...]` to only check for that file `in` that set of model versions, or to exclude (`except`) that file from being check for a set of model versions.

Example

```
comp:
        actual:
                files:
                    - "bin/fesom*"
                    - "bin/oifs"
                    - "bin/rnfma"
        check: {}
run:
        actual:
                errors:
                    - "MISSING FILES"
                files:
                    - "restart/fesom/fesom.*.oce.restart/hnode.nc*"
                    - "restart/fesom/fesom.*.oce.restart/salt.nc*"
                    - "restart/fesom/fesom.*.oce.restart/ssh_rhs_old.nc*"
                    - "restart/fesom/fesom.*.oce.restart/temp.nc*"
                    - "restart/fesom/fesom.*.oce.restart/urhs_AB.nc*"
                    - "restart/fesom/fesom.*.oce.restart/vrhs_AB.nc*"
                    - "restart/fesom/fesom.*.oce.restart/w_impl.nc*"
                    - "restart/fesom/fesom.*.ice.restart/area.nc*"
                    - "restart/fesom/fesom.*.ice.restart/hice.nc*"
                    - "restart/fesom/fesom.*.ice.restart/hsnow.nc*"
                    - "restart/fesom/fesom.*.ice.restart/ice_albedo.nc*"
                    - "restart/fesom/fesom.*.ice.restart/ice_temp.nc*"
                    - "restart/fesom/fesom.*.ice.restart/uice.nc*"
                    - "restart/fesom/fesom.*.ice.restart/vice.nc*"
                    - "restart/oasis3mct/rmp_*"
                    - "restart/oasis3mct/rstas.nc*"
                    - "restart/oasis3mct/rstos.nc*"
                    - "restart/oifs/*/BLS*"
                    - "restart/oifs/*/LAW*"
                    - "restart/oifs/*/rcf"
                    - "restart/oifs/*/srf*"
                    - "restart/oifs/*/waminfo*"
                    - "outdata/oifs/*/ICMGG* except [frontiers-xios, v3.1]"
                    - "outdata/oifs/*/ICMSH* except [frontiers-xios, v3.1]"
                    - "outdata/oifs/*/ICMUA* except [frontiers-xios, v3.1]"
                    - "outdata/oifs/atm_remapped* in [frontiers-xios, v3.1]"
                    - "outdata/fesom/*.fesom.*.nc"
        check: {}
computers:
        - ollie
        - mistral
        - juwels
        - aleph
```

(continues on next page)

(continued from previous page)

```
      - blogin
      - levante
```

16.7 Local test configuration (`test_config.yaml`)

File location: `esm_tools/src/esm_tests/test_config.yaml` **Versioned**: No, user specific, git-ignored

This file gives you control on which tests `esm_tests` will run in the current machine, independently of what tests are defined in the *Model control files*. The current machine needs to be included in the *Model control file* for the test to run (i.e. `test_config.yaml` runs only the tests included there and supported on the current platform). The syntax is as follows:

```
<model1>:
    - <runscript1_name>.yaml
    - <runscript2_name>.yaml
    - [ ... ]
<model2>: all
[ ... ]
```

The `model` sections need to be named after the folders in `esm_tools/src/esm_tests/resources/runscripts`. If you want to run all the suported runscripts for a model in this platform, make the `model` section have the value `all`. If you want to select a set of **supported runscripts** make the `model` be a list of runscripts (this runscripts need to be in `esm_tools/src/esm_tests/resources/runscripts/<model>/`). If you want to run all the supported runscripts for all supported models in this platform, but still keep this file around (i.e. commented most of the contents), make the file content be an empty dictionary (`{}`).

Example

```
#{}
awiesm: #all
    - all_awiesm-2.1-recom.yaml
#    - awiesm-2.1_icebergs.yaml
    - bootstrap.yaml
    - pico.yaml
    - PI_ctrl_awiesm-2.1-wiso.yaml
    - pi.yaml
    - pi-wiso.yaml
echam: all
fesom: all
awicm: all
#    - awicm1-CMIP6-initial-monthly.yaml
#    - awicm2-initial-monthly.yaml
fesom-recom:
   - fesom-recom1.4-initial-daily.yaml
awicm3: all
#    - awicm3-v3.1-TCO95L91-CORE2_initial
#    - awicm3-frontiers-TCO159L91-CORE2_initial.yaml
#oifsamip: all
#vilma-pism: all
```

16.8 ESM-Tests cookbook

16.8.1 How to include a new model/runscript

1. Add the given runscript to `esm_tools/src/esm_tests/resources/runscripts/<model>/`
2. Make sure your runscript has a meaningful name
3. Make sure your runscript has the correct model `version` defined, for a standalone model in the section of the model (not in `general`), and for a coupled setup, both in the `general` section and in the coupled setup section (e.g. `awiesm` section). This version will be used by *ESM-Test* for the `esm_master` command to compile
4. Modify the following variables to take the environment variables setup by *ESM-Tests*:

```
general:
    account: !ENV ${ACCOUNT}
    base_dir: !ENV ${ESM_TESTING_DIR}
<standalone-model/setup>:
    model_dir: !ENV ${MODEL_DIR}
```

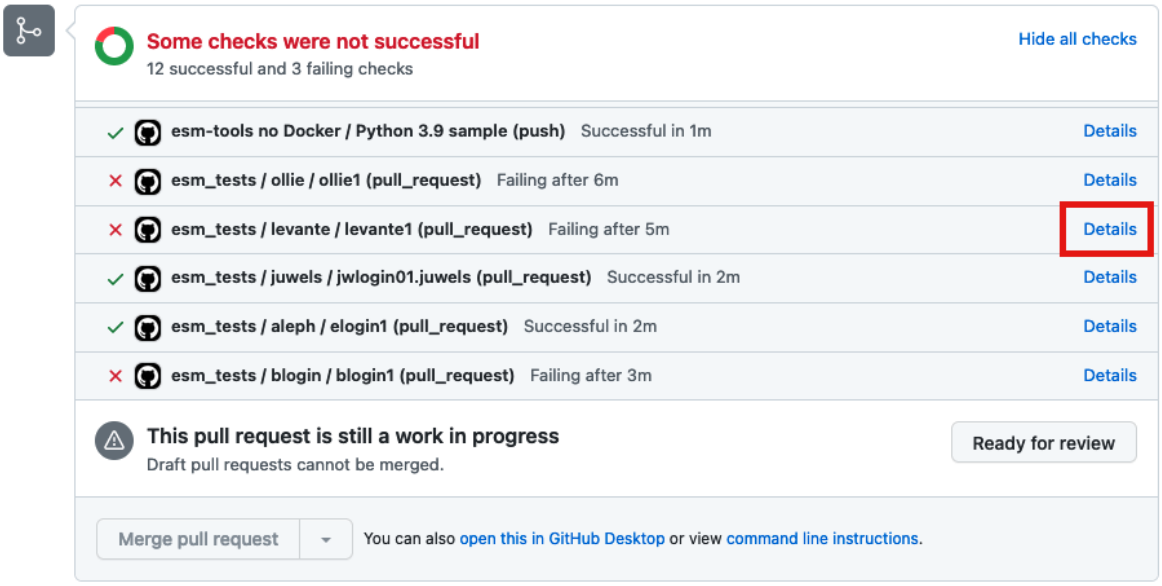
5. Generalize the runscript to be able to run in the computers where you'd want it to be supported (i.e. add the necessary `choose_computer.name` switches)
6. Create the *Model control file* (`esm_tools/src/esm_tests/resources/runscripts/<model>/config.yaml`). See ref:*esm_tests:Model control file (``config.yaml``)* for details about the content
7. If you are using the Local test configuration (``test_config.yaml``) to exclude some models, make sure the current model is included, so that your tests can be run locally.

16.8.2 How to include a new platform for in an existing model

1. In the corresponding *Model control file* (`esm_tools/src/esm_tests/resources/runscripts/<model>/config.yaml`), add the name of the platform to the `computers` list
2. In the runscripts (`esm_tools/src/esm_tests/resources/runscripts/<model>/<runscript>.yaml`), add the necessary case to the `choose_computer.name` to specify pool directories, forcing files, `nproc`, etc.

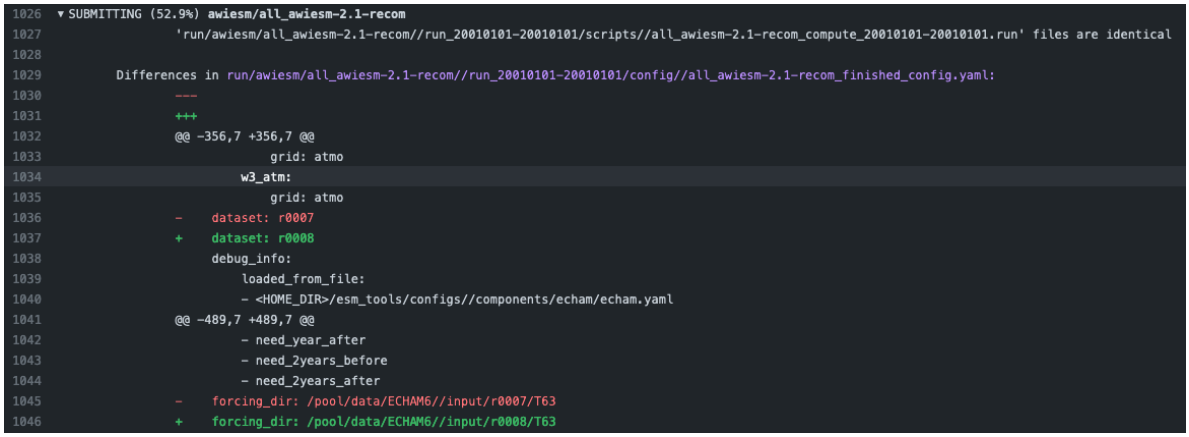
16.8.3 How to approve changes on a GitHub Pull-Request

1. In the pull-request, if all the tests passed you don't need to approve any changes, you can jump directly to step 4.
2. If any of the tests labelled as `esm_tests` failed (click on the triangles to expand screen captures):




```
awiesm:
|   2.1:
|       awiesm-2.1_icebergs:
|           ollie:      comp files identical    run files differ
|       pico:
|           ollie:      comp files identical    run files differ
|       bootstrap:
|           ollie:      comp files identical    run files differ
|       pi:
|           ollie:      comp files identical    run files differ
|       pi-wiso:
|           ollie:      comp files identical    run files differ
|   2.1-wiso:
|       PI_ctrl_awiesm-2.1-wiso:
|           ollie:      comp files identical    run files differ
|   2.1-recom:
|       all_awiesm-2.1-recom:
|           ollie:      comp files identical    run files differ
```

```
  52 ▸ COMPILING (5.9%) vilma-pism-1.0v1d
  57 ▸ COMPILING (11.8%) fesom-recom-1.4
  62 ▸ COMPILING (17.6%) fesom-2.0
  66 ▸ COMPILING (23.5%) fesom-2.1
  70 ▸ COMPILING (29.4%) echam-6.3.04p1
  74 ▸ COMPILING (35.3%) awiesm-2.1
  80 ▸ COMPILING (41.2%) awiesm-2.1
  86 ▸ COMPILING (47.1%) awiesm-2.1-wiso
  92 ▸ COMPILING (52.9%) awiesm-2.1
  98 ▸ COMPILING (58.8%) awiesm-2.1
 104 ▸ COMPILING (64.7%) awiesm-2.1
 110 ▸ COMPILING (70.6%) awiesm-2.1-recom
 116 ▸ COMPILING (76.5%) awicm-CMIP6
 122 ▸ COMPILING (82.4%) awicm-2.0
 128 ▸ COMPILING (88.2%) awicm3-v3.0
 135 ▸ COMPILING (94.1%) awicm3-frontiers-xios
 143 ▸ COMPILING (100.0%) awicm3-v3.1
 151 ▸ SUBMITTING (5.9%) vilma-pism/vilma1d_1ky-pism_1y
 152 ▸ SUBMITTING (11.8%) fesom-recom/fesom-recom1.4-initial-daily
 161 ▸ SUBMITTING (17.6%) fesom/fesom2.0-initial-monthly
 169 ▸ SUBMITTING (23.5%) fesom/fesom2.1-initial-monthly
 179 ▸ SUBMITTING (29.4%) echam/echam6.3.04p1-initial-monthly
 230 ▸ SUBMITTING (35.3%) awiesm/awiesm-2.1_icebergs
 372 ▸ SUBMITTING (41.2%) awiesm/pico
 514 ▸ SUBMITTING (47.1%) awiesm/PI_ctrl_awiesm-2.1-wiso
 662 ▸ SUBMITTING (52.9%) awiesm/bootstrap
 805 ▸ SUBMITTING (58.8%) awiesm/pi
 947 ▸ SUBMITTING (64.7%) awiesm/pi-wiso
1095 ▸ SUBMITTING (70.6%) awiesm/all_awiesm-2.1-recom
1238 ▸ SUBMITTING (76.5%) awicm/awicm1-CMIP6-initial-monthly
1295 ▸ SUBMITTING (82.4%) awicm/awicm2-initial-monthly
1352 ▸ SUBMITTING (88.2%) awicm3/awicm3-v3.0-TCO159L91-CORE2_initial
1364 ▸ SUBMITTING (94.1%) awicm3/awicm3-frontiers-xios-TCO159L91-CORE2_initial
1376 ▸ SUBMITTING (100.0%) awicm3/awicm3-v3.1-TCO95L91-CORE2_initial
```



3. If there are no problematic differences, and the pull-request has been already reviewed and is just ready to be merged, write a message on the PR containing `#approve-changes`. This will commit the new files from the tests as the `last-state`, in the `esm_tests_info` repository.

> **Warning:** Currently, `#approve-changes` does not update the test status on GitHub, once the operation finishes. If you want to see whether `#approve_changes` finished or not you have to navigate to the `Actions` tab in GitHub. If you want to see all tests green, wait until `#approve-changes` finishes, and relaunch the tests for the last failed set of tests in the PR. Miguel - I know this is a pain, but I could not figure out how to do all this automatically (I wasted enough time on GitHub Actions for years to come).

4. Bump the version and wait that the bumpversion commit shows up.
5. You can now merge.

CHAPTER

SEVENTEEN

ESM MOTD

The package `esm_motd` is an *ESM-Tools* integrated *message-of-the-day* system, intended as a way for the *ESM-Tools Development Team* to easily announce new releases and bug fixes to the users without the need of emailing.

It checks the versions of the different *ESM-Tools* packages installed by the user, and reports back to the user (writing to *stdout*) about packages that have available updates, and what are the new improvements that they provide (i.e. reports back that a bug in a certain package has been solved).

This check occurs every time the user uses `esm_runscripts`.

The messages, their corresponding versions and other related information is stored online in GitHub and accessed by *ESM-Tools* also online to produce the report. The user can look at this file if necessary here: *<https://github.com/esm-tools/esm_tools/tree/release/esm_tools/motd/motd.yaml>_*.

Warning: The `motd.yaml` file is to be modified exclusively by the ESM-Tools Core Development Team, so… stay away from it ;-)

CHAPTER

EIGHTEEN

COOKBOOK

In this chapter you can find multiple recipes for different ESM-Tools functionalities, such running a model, adding forcing files, editing defaults in namelists, etc.

If you'd like to contribute with your own recipe, or ask for a recipe, please open a documentation issue on our GitHub repository.

Note: Throughout the cookbook, we will sometimes refer to a nested part of a configuration via dot notation, e.g. `a.b.c`. Here, we mean the following in a YAML config file:

```yaml
a:
  b:
    c: "foo"
```

This would indicate that the value of `a.b.c` is `"foo"`. In Python, you would access this value as `a["b"]["c"]`.

18.1 Change/Add Flags to the sbatch Call

Feature available since version: 4.2

If you are using *SLURM* batch system together with *ESM-Tools* (so far the default system), you can modify the `sbatch` call flags by modifying the following variables from your runscript, inside the `computer` section:

Key	Description
mail_type, mail_user	Define these two variables to get updates about your slurm-job through email.
single_proc_submit_flag	By default defined as `--ntasks-per-node=1`
additional_flags	To add any additional flag that is not predefined in *ESM-Tools*

18.1.1 Example

Assume you want to run a simulation using the Quality of Service flag (`--qos`) of *SLURM* with value `24h`. Then, you'll need to define the `additional_flags` inside the `computer` section of your runscript. This can be done by adding the following to your runscript:

```
computer:
    additional_flags: "--qos=24h"
```

18.1.2 Adding more than one flag

Alternatively, you can include a list of additional flags:

```
computer:
    additional_flags:
        - "--qos=24h"
        - "--comment='My Slurm Comment'"
```

See the documentation for the batch scheduler on your HPC system to see the allowed options.

18.2 Applying a temporary disturbance to ECHAM to overcome numeric instability (lookup table overflows of various kinds)

Feature available since version: `esm_runscripts v4.2.1`

From time to time, the `ECHAM` family of models runs into an error resulting from too high wind speeds. This may look like this in your log files:

```
30: ================================================================================
30:
30: FATAL ERROR in cuadjtq (1):  lookup table overflow
30:  FINISH called from PE: 30
```

To overcome this problem, you can apply a small change to the factor "by which stratospheric horizontal diffussion is increased from one level to the next level above." (`mo_hdiff.f90`), that is the namelist parameter `enstdif`, in the `dynctl` section of the `ECHAM` namelist. As this is a common problem, there is a way to have the run do this for specific years of your simulation. Whenever a model year crashes due to numeric instability, you have to apply the method outlined below.

1. Generate a file to list years you want disturbed.

   In your experiment script folder (**not** the one specific for each run), you can create a file called `disturb_years.dat`. An abbreviated file tree would look like:

2. Add years you want disturbed.

   The file should contain a list of years the disturbance should be applied to, seperated by new lines. In practice, you will add a new line with the value of the model year during which the model crashes whenever such a crash occurs.

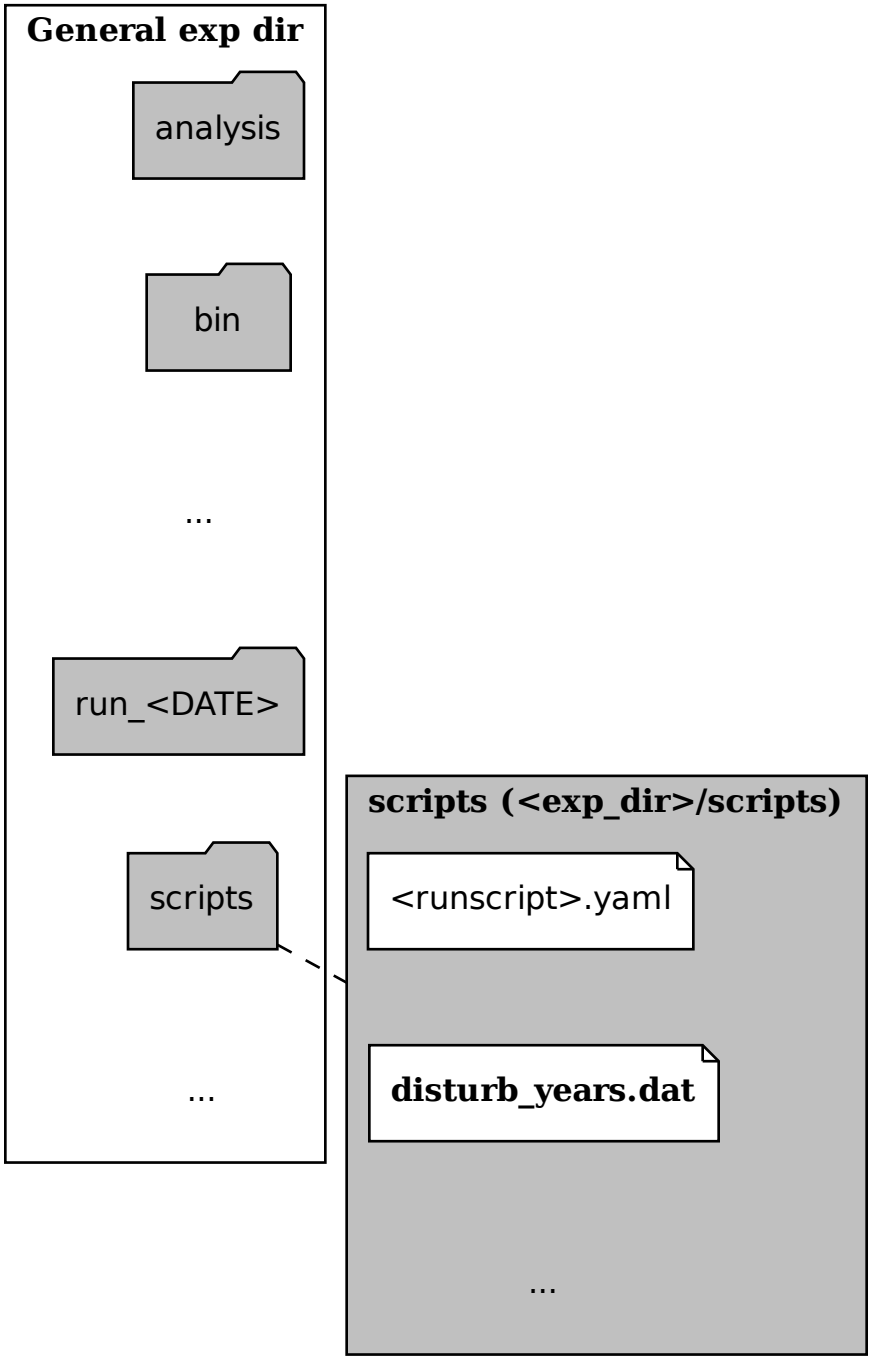


Fig. 1: disturb_years.dat location

18.2.1 Example

In this example, we disturb the years 2005, 2007, and 2008 of an experiment called `EXAMPLE` running on `ollie`:

```
$ cat /work/ollie/pgierz/test_esmtools/EXAMPLE/scripts/disturb_years.dat
2005
2007
2008
```

You can also set the disturbance strength in your configuration under `echam.disturbance`. The default is `1.000001`. Here, we apply a 200% disturbance whenever a "disturb_year" is encountered.

```
echam:
    disturbance: 2.0
```

18.2.2 See also

- ECHAM6 User Handbook, Table 2.4, dynctl
- Relevant source code

18.3 Changing Namelist Entries from the Runscript

Feature available since version: 4.2

You can modify namelists directly from your user yaml runscript configuration.

1. Identify which namelist you want to modify and ensure that it is in the correct section. For example, you can only modify `ECHAM` specific namelists from an `ECHAM` block.
2. Find the subsection ("chapter") of the namelist you want to edit.
3. Find the setting ("key") you want to edit
4. Add a `namelist_changes` block to your configuration, specify next the namelist filename you want to modify, then the chapter, then the key, and finally the desired value.

In dot notation, this will look like: `<model_name>.namelist_changes.<namelist_name>.<chapter_name>.<key_name> = <value>`

18.3.1 Example

Here are examples for just the relevant YAML change, and for a full runscript using this feature.

Snippet

Full Runscript

In this example, we modify the `co2vmr` of the `radctl` section of `namelist.echam`.

```
echam:
    namelist_changes:
        namelist.echam:
            radctl:
                co2vmr: 1200e-6
```

In this example, we set up AWI-ESM 2.1 for a 4xCO2 simulation. You can see how multiple namelist changes are applied in one block.

```
general:
    setup_name: "awiesm"
    compute_time: "02:30:00"
    initial_date: "2000-01-01"
    final_date: "2002-12-31"
    base_dir: "/work/ab0246/a270077/For_Christian/experiments/"
    nmonth: 0
    nyear: 1
    account: "ab0246"

echam:
    restart_unit: "years"
    nprocar: 0
    nprocbr: 0
    namelist_changes:
            namelist.echam:
                    radctl:
                            co2vmr: 1137.e-6
                    parctl:
                            nprocar: 0
                            nprocbr: 0
                    runctl:
                            default_output: True

awiesm:
    version: "2.1"
    postprocessing: true
    scenario: "PALEO"
    model_dir: "/work/ab0246/a270077/For_Christian/model_codes/awiesm-2.1/"

fesom:
    version: "2.0"
    res: "CORE2"
    pool_dir: "/pool/data/AWICM/FESOM2"
    mesh_dir: "/work/ba1066/a270061/mesh_CORE2_finaltopo_mean/"
    restart_rate: 1
    restart_unit: "y"
    restart_first: 1
    lresume: 0
    namelist_changes:
        namelist.config:
            paths:
                ClimateDataPath: "/work/ba0989/a270077/AWIESM_2_1_LR_concurrent_rad/
↪nonstandard_input_files/fesom/hydrography/"

jsbach:
    input_sources:
        jsbach_1850: "/work/ba1066/a270061/mesh_CORE2_finaltopo_mean/tarfilesT63/input/
↪jsbach/jsbach_T63CORE2_11tiles_5layers_1850.nc"
```

18.3.2 Practical Usage

It is generally a good idea to run your simulation once in **check** mode before actually submitting and examining the resulting namelists:

```
$ esm_runscripts <your_config.yaml> -e <expid> -c
```

The namelists are printed in their final form as part of the log during the job submission and can be seen on disk in the `work` folder of your first `run_XZY` folder.

Note that you can have several chapters for one namelist or several namelists included in one `namelist_changes` block, but you can only have one `namelist_changes` block per model or component (see *Changing Namelists*).

18.3.3 Unusual Namelists

Some times, you have strange namelists of the form:

```
sn_tracer(1)   = 'DET'   , 'Detritus                    '  , 'mmole-N/m3' ,  .false.
sn_tracer(2)   = 'ZOO'   , 'Zooplankton concentration  '  , 'mmole-N/m3' ,  .false.
sn_tracer(3)   = 'PHY'   , 'Phytoplankton concentration'  , 'mmole-N/m3' ,  .false.
```

To correctly insert this via `esm-tools`, you can use:

```
namelist_changes:
      namelist_top_cfg:
        namtrc:
           sn_tracer: "remove_from_namelist"
           sn_tracer(1)%clsname: DET
           sn_tracer(2)%clsname: ZOO
           sn_tracer(3)%clsname: PHY
           sn_tracer(1)%cllname: "Detritus"
           sn_tracer(2)%cllname: "Zooplankton concentration"
           sn_tracer(3)%cllname: "Phytoplankton concentration"
           sn_tracer(1:3)%clunit: "mmole-N/m3"
```

18.3.4 See also

- Default namelists on GitHub
- *Changing Namelists*
- *What Is YAML?*

18.4 Heterogeneous Parallelization Run (MPI/OpenMP)

Feature available since version: 5.1

In order to run a simulation with hybrid MPI/OpenMP parallelization include the following in your runscript:

1. Add `heterogenous_parallelization: true` in the `computer` section of your runscript. If the `computer` section does not exist create one.
2. Add `omp_num_threads: <number>` to the sections of the components you'd like to have OpenMP parallelization.

18.4.1 Example

AWICM3

In *AWICM3* we have 3 components: *FESOM-2*, *OpenIFS* and *RNFMAP*. We want to run *OpenIFS* with 8 OpenMP threads, *RNFMAP* with 48, and *FESOM-2* with 1. Then, the following lines need to be added to our runscript:

```
general:
    [ ... ]
computer:
    heterogeneous_parallelization: true
    [ ... ]
fesom:
    omp_num_threads: 1
    [ ... ]
oifs:
    omp_num_threads: 8
    [ ... ]
rnfmap:
    omp_num_threads: 48
    [ ... ]
```

18.4.2 See also

- esm_variables:Runtime variables

18.5 How to setup runscripts for different kind of experiments

This recipe describes how to setup a runscript for the following different kinds of experiments. Besides the variables described in *ESM-Tools Variables*, add the following variables to your runscript, as described below.

- **Initial run**: An experiment from initial model conditions.

```
general:
    lresume: 0
```

- **Restart**: An experiment that restarts from a previous experiment with the same experiment id.

```
general:
    lresume: 1
```

- **Branching off**: An experiment that restarts from a previous experiment but with a different experiment id.

```
general:
    lresume: 1
    ini_parent_exp_id: <old-experiment-id>
    ini_restart_dir: <path-to-restart-dir-of-old-experiment>/restart/
```

- **Branching off and redate**: An experiment that restarts from a previous experiment with a different experiment id and if this experiment should be continued with a diiferent start date.

```
general:
    lresume: 1
    ini_parent_exp_id: <old-experiment-id>
    ini_restart_dir: <path-to-restart-dir-of-old-experiment>/restart/
    first_initial_year: <year>
```

18.5.1 See also

- *ESM-Tools Variables*
- *What Is YAML?*

18.6 Implement a New Model

Feature available since version: 4.2

Note: since version 6.20.2 a template is available in `esm_tools/configs/templates/component_template.yaml`

---

1. Upload your model into a repository such us *gitlab.awi.de*, *gitlab.dkrz.de* or *GitHub*. Make sure to set up the right access permissions, so that you comply with the licensing of the software you are uploading.
2. If you are interested in implementing more than one version of the model, we recommend you to commit them to the master branch in the order they were developed, and that you create a tag per version. For example:
   a. Clone the empty master branch you just created and add your model files to it:

   ```
   $ git clone https://<your_repository>
   $ cp -rf <your_model_files_for_given_version> <your_repository_folder>
   $ git add .
   ```

   b. Commit, tag the version and push the changes to your repository:

   ```
   $ git commit -m "your comment here"
   $ git tag -a <version_id> -m "your comment about the version"
   $ git push -u origin <your_master_branch>
   $ git push origin <version_id>
   ```

   c. Repeat steps *a* and *b* for all the versions that you would like to be present in ESM-Tools.
3. Now that you have your model in a repository you are ready to implement it into *esm_tools*. First, you will need to create your own branch of *esm_tools*, following the steps 1-4 in *Contribution to esm_tools Package*. The recommended name for the branch would be `feature/<name_of_your_model>`.
4. Then you will need to create a folder for your model inside `esm_tools/configs/components` and create the model's *yaml* file:

   ```
   $ mkdir <PATH>/esm_tools/configs/components/<model>
   $ touch <PATH>/esm_tools/configs/components/<model>/<model>.yaml
   ```

5. Use your favourite text editor to open and edit your `<model>.yaml` in the `esm_tools/configs/components/<model>` folder:

```
$ <your_text_editor> <PATH>/esm_tools/configs/components/<model>/<model>.yaml
```

6. Complete the following information about your model:

```
# YOUR_MODEL YAML CONFIGURATION FILE
#

model: your_model_name
type: type_of_your_model       # atmosphere, ocean, etc.
version: "the_default_version_of_your_model"
```

7. Include the names of the different versions in the `available_versions` section and the compiling information for the default version:

```
[...]

available_versions:
- "1.0.0"
- "1.0.1"
- "1.0.2"
git-repository: "https://your_repository.git"
branch: your_model_branch_in_your_repo
install_bins: "path_to_the_binaries_after_comp"
comp_command: "your_shell_commands_for_compiling"      # You can use the defaults "$
↪{defaults.comp_command}"
clean_command: "your_shell_commands_for_cleaning"      # You can use the defaults "$
↪{defaults.clean_command}"

executable: your_model_command

setup_dir: "${model_dir}"
bin_dir: "${setup_dir}/name_of_the_binary"
```

   In the `install_bins` key you need to indicate the path inside your model folder where the binaries are compiled to, so that *esm_master* can find them once compiled. The `available_versions` key is needed for *esm_master* to list the versions of your model. The `comp_command` key indicates the command needed to compile your model, and can be set as `${defaults.comp_command}` for a default command (`mkdir -p build; cd build; cmake ..; make install -j `nproc --all``), or you can define your own list of compiling commands separated with `;` (`"command1; command2"`).

8. At this point you can choose between including all the version information inside the same `<model>.yaml` file, or to distribute this information among different version files:

   Single file

   Multiple version files

   In the `<model>.yaml`, use a `choose_` switch (see *Switches (choose_)*) to modify the default information that you added in step 7 to meet the requirements for each specific version. For example, each different version has its own git branch:

```
choose_version:
        "1.0.0":
                branch: "1.0.0"
        "1.0.1":
```

(continues on next page)

(continued from previous page)

```
            branch: "1.0.1"
    "1.0.2":
            branch: "develop"
```

a. Create a *yaml* file per version or group of versions. The name of these files should be the same as the ones in the `available_versions` section, in the main `<model>.yaml` file or, in the case of a file containing a group of versions, the shared name among the versions (i.e. `fesom-2.0.yaml`):

```
$ touch <PATH>/esm_tools/configs/<model>/<model-version>.yaml
```

b. Open the version file with your favourite editor and include the version specific changes. For example, you want that the version `1.0.2` from your model pulls from the `develop` git branch, instead of from the default branch. Then you add to the `<model>-1.0.2.yaml` version file:

```
branch: "develop"
```

Another example is the `fesom-2.0.yaml`. While `fesom.yaml` needs to contain all `available_versions`, the version specific changes are split among `fesom.yaml` (including information about versions 1) and `fesom-2.0.yaml` (including information about versions 2):

fesom.yaml

fesom-2.0.yaml

```
[ ... ]

available_versions:
- '2.0-o'
- '2.0-esm-interface'
- '1.4'
- '1.4-recom'
- '1.4-recom-awicm'
- '2.0-esm-interface-yac'
- '2.0-paleodyn'
- '2.0'
- '2.0-r' # OG: temporarily here
choose_version:
  '1.4-recom-awicm':
    branch: fesom_recom_1.4_master
    destination: fesom-1.4
  '1.4-recom':
    branch: fesom_recom_1.4_master
    destination: fesom-1.4

[ ... ]
```

```
[ ... ]

choose_version:
  '2.0':
    branch: 2.0.2
    git-repository:
    - https://gitlab.dkrz.de/FESOM/fesom2.git
```

(continues on next page)

(continued from previous page)

```
    - github.com/FESOM/fesom2.git
    install_bins: bin/fesom.x
  2.0-esm-interface:
    branch: fesom2_using_esm-interface
    destination: fesom-2.0
    git-repository:
    - https://gitlab.dkrz.de/a270089/fesom-2.0_yac.git
    install_bins: bin/fesom.x

[ ... ]
```

Note: These are just examples of model configurations, but the parser used by *ESM-Tools* to read *yaml* files (*esm_parser*) allows for a lot of flexibility in their configuration; i.e., imagine that the different versions of your model are in different repositories, instead of in different branches, and their paths to the binaries are also different. Then you can include the `git-repository` and `install_bins` variables inside the corresponding version case for the `choose_version`.

9. You can now check if *esm_master* can list and install your model correctly:

```
$ esm_master
```

This command should return, without errors, a list of available models and versions including yours. Then you can actually try installing your model in the desired folder:

```
$ mkdir ~/model_codes
$ cd ~/model_codes
$ esm_master install-your_model-version
```

10. If everything works correctly you can check that your changes pass `flake8`:

```
$ flake8 <PATH>/esm_tools/configs/components/<model>/<model>.yaml
```

Use this link to learn more about `flake8` and how to install it.

11. Commit your changes, push them to the `origin` remote repository and submit a pull request through GitHub (see steps 5-7 in *Contribution to esm_tools Package*).

Note: You can include all the compiling information inside a `compile_infos` section to avoid conflicts with other `choose_version` switches present in your configuration file.

18.6.1 See also

- *ESM-Tools Variables*
- *Switches (choose_)*
- *What Is YAML?*

18.7 Implement a New Coupled Setup

Feature available since version: 4.2

An example of the different files needed for *AWICM* setup is included at the end of this section (see recipes/add_model_setup:Example).

1. Make sure the models, couplers and versions you want to use, are already available for *esm_master* to install them (`$ esm_master` and check the list). If something is missing you will need to add it following the instructions in *Implement a New Model*.
2. Once everything you need is available to *esm_master*, you will need to create your own branch of *esm_tools*, following the steps 1-4 in *Contribution to esm_tools Package*.
3. Setups need two types of files: 1) **coupling files** containing information about model versions and coupling changes, and 2) **setup files** containing the general information about the setup and the model changes. In this step we focus on the creation of the **coupling files**.
   a. Create a folder for your couplings in `esm_tools/configs/couplings`:

   ```
   $ cd esm_tools/configs/couplings/
   $ mkdir <coupling_name1>
   $ mkdir <coupling_name2>
   ...
   ```

   The naming convention we follow for the coupling files is `component1-version+component2-version+...`

   b. Create a *yaml* file inside the coupling folder with the same name:

   ```
   $ touch <coupling_name1>/<coupling_name1>.yaml
   ```

   c. Include the following information in each coupling file:

   ```yaml
   components:
   - "model1-version"
   - "model2-version"
   - [ ... ]
   - "coupler-version"
   coupling_changes:
   - sed -i '/MODEL1_PARAMETER/s/OFF/ON/g' model1-1.0/file_to_change
   - sed -i '/MODEL2_PARAMETER/s/OFF/ON/g' model2-1.0/file_to_change
   - [ ... ]
   ```

   The `components` section should list the models and couplers used for the given coupling including, their required version. The `coupling_changes` subsection should include a list of commands to make the necessary changes in the component's compilation configuration files (`CMakeLists.txt`, `configure`, etc.), for a correct compilation of the coupled setup.
4. Now, it is the turn for the creation of the **setup file**. Create a folder for your coupled setup inside `esm_tools/configs/setups` folder, and create a *yaml* file for your setup:

   ```
   $ mkdir <PATH>/esm_tools/configs/setups/<your_setup>
   $ touch <PATH>/esm_tools/configs/setups/<your_setup>/<setup>.yaml
   ```
5. Use your favourite text editor to open and edit your `<setup>.yaml` in the `esm_tools/configs/setups/<your_setup>` folder:

```
$ <your_text_editor> <PATH>/esm_tools/configs/setups/<your_setup>/<setup>.yaml
```

6. Complete the following information about your setup:

```
##############################################################################
#####
######################### NAME_VERSION YAML CONFIGURATION FILE ###################
#####
##############################################################################
#####

general:
        model: your_setup
        version: "your_setup_version"

        coupled_setup: True

        include_models:           # List of models, couplers and componentes of the␣
↪setup.
                - component_1     # Do not include the version number
                - component_2
                - [ ... ]
```

Note: *Models* do not have a `general` section but in the *setups* the `general` section is mandatory.

7. Include the names of the different versions in the `available_versions` section:

```
general:

        [ ... ]

        available_versions:
                - "1.0.0"
                - "1.0.1"
```

The `available_versions` key is needed for *esm_master* to list the versions of your setup.

8. In the `<setup>.yaml`, use a `choose_` switch (see *Switches (choose_)*) to assign the coupling files (created in step 3) to their corresponding setup versions:

```
general:

    [ ... ]

    choose_version:
            "1.0.0":
                    couplings:
                            - "model1-1.0+model2-1.0"
            "1.0.1":
                    couplings:
                            - "model1-1.1+model2-1.1"

    [ ... ]
```

9. You can now check if *esm_master* can list and install your coupled setup correctly:

```
$ esm_master
```

This command should return, without errors, a list of available setups and versions including yours. Then you can actually try installing your setup in the desire folder:

```
$ mkdir ~/model_codes
$ cd ~/model_codes
$ esm_master install-your_setup-version
```

10. If everything works correctly you can check that your changes pass `flake8`:

```
$ flake8 <PATH>/esm_tools/configs/setups/<your_setup>/<setup>.yaml
$ flake8 <PATH>/esm_tools/configs/couplings/<coupling_name>/<coupling_name>.yaml
```

Use this link to learn more about `flake8` and how to install it.

11. Commit your changes, push them to the `origin` remote repository and submit a pull request through GitHub (see steps 5-7 in *Contribution to esm_tools Package*).

18.7.1 Example

Here you can have a look at relevant snippets of some of the *AWICM-1.0* files.

fesom-1.4+echam-6.3.04p1.yaml

awicm.yaml

One of the coupling files for *AWICM-1.0* (`esm_tools/configs/couplings/fesom-1.4+echam-6.3.04p1/fesom-1.4+echam-6.3.04p1.yaml`):

```
components:
- echam-6.3.04p1
- fesom-1.4
- oasis3mct-2.8
coupling_changes:
- sed -i '/FESOM_COUPLED/s/OFF/ON/g' fesom-1.4/CMakeLists.txt
- sed -i '/ECHAM6_COUPLED/s/OFF/ON/g' echam-6.3.04p1/CMakeLists.txt
```

Setup file for *AWICM* (`esm_tools/configs/setups/awicm/awicm.yaml`):

```
#############################################################################################
######################### AWICM 1 YAML CONFIGURATION FILE ###################################
#############################################################################################



general:
        model: awicm
        #model_dir: ${esm_master_dir}/awicm-${version}

        coupled_setup: True

        include_models:
```

(continues on next page)

(continued from previous page)

```
            - echam
            - fesom
            - oasis3mct

    version: "1.1"
    scenario: "PI-CTRL"
    resolution: ${echam.resolution}_${fesom.resolution}
    postprocessing: false
    post_time: "00:05:00"
    choose_general.resolution:
            T63_CORE2:
                    compute_time: "02:00:00"
            T63_REF87K:
                    compute_time: "02:00:00"
            T63_REF:
                    compute_time: "02:00:00"
    available_versions:
    - '1.0'
    - '1.0-recom'
    - CMIP6
    choose_version:
      '1.0':
        couplings:
        - fesom-1.4+echam-6.3.04p1
      '1.0-recom':
        couplings:
        - fesom-1.4+recom-2.0+echam-6.3.04p1
      CMIP6:
        couplings:
        - fesom-1.4+echam-6.3.04p1
```

18.7.2 See also

- *ESM-Tools Variables*
- *Switches (choose_)*
- *What Is YAML?*

18.8 Implement a New HPC Machine

To implement a new HPC machine to *ESM-Tools*, two files need to be updated and created, respectively:

- `<PATH>/esm_tools/configs/machines/all_machines.yaml`
- `<PATH>/esm_tools/configs/machines/<new_machine>.yaml`

1. Add an additional entry for the new machine.

   Use your favourite text editor and open the file `<PATH>/esm_tools/configs/machines/all_machines.yaml`:

```
$ <your_text_editor> <PATH>/esm_tools/configs/machines/all_machines.yaml
```

and add a new entry for the new machine (replace placeholders indicated by <...>)

```
<new_machine>:
   login_nodes: '<hostname>*' # A regex pattern that matches the hostname of login␣
↪nodes
   compute_nodes: '<compute_notes>' # A regex pattern that matches the hostname of␣
↪compute nodes
```

2. Create a new machine file.

   Use your favourite text editor to create and edit a new machine file <new_machine>.yaml in the esm_tools/configs/machines/ folder:

```
$ <your_text_editor> <PATH>/esm_tools/configs/machines/<new_machine>.yaml
```

   A template file (machine_template.yaml) is available in configs/templates, so you can alternatively copy this file into the configs/machines folder edit the relevant entries:

```
$ cp <PATH>/esm_tools/configs/templates/machine_template.yaml  <PATH>/esm_tools/
↪configs/machines/<new_machine>.yaml
$ <your_text_editor> <PATH>/esm_tools/configs/machines/<new_machine>.yaml
```

   You can also reproduce the two steps above simply by running the following esm_tools command:

```
$ esm_tools create-new-config <PATH>/esm_tools/configs/machines/<new_machine>.yaml -
↪t machine
```

   This will copy the machine_template.yaml in the target location and open the file in your default editor.

18.8.1 See also

- *ESM-Tools Variables*
- *Switches (choose_)*
- *What Is YAML?*

18.9 Include a New Forcing/Input File

Feature available since version: 4.2

There are several ways of including a new forcing or input file into your experiment depending on the degree of control you'd like to achieve. An important clarification is that <forcing/input>_sources file dictionary specifies the **sources** (paths to the files in the pools or personal folders, that need to be copied or linked into the experiment folder). On the other hand <forcing/input>_files specifies which of these sources are to be **included in the experiment**. This allows us to have many sources already available to the user, and then the user can simply choose which of them to use by chosing from <forcing/input>_files. <forcing/input>_in_work is used to copy the files into the work folder (<base_dir>/<exp_id>/run_<DATE>/work) if necessary and change their name. For more technical details see *File Dictionaries*.

The next sections illustrate some of the many options to handle forcing and input files.

18.9.1 Source Path Already Defined in a Config File

1. Make sure the source of the file is already specified inside the `forcing_sources` or `input_sources` *file dictionaries* in the configuration file of the setup or model you are running, or on the `further_reading` files.
2. In your runscript, include the *key* of the source file you want to include inside the `forcing_files` or `input_files` section.

Note: Note that the *key* containing the source in the `forcing_sources` or `input_sources` can be different than the key specified in `forcing_files` or `input_files`.

Example

ECHAM

In ECHAM, the source and input file paths are specified in a separate file (`<PATH>/esm_tools/configs/components/echam/echam.datasets.yaml`) that is reached through the `further_reading` section of the `echam.yaml`. This file includes a large number of different sources for input and forcing contained in the pool directories of the HPC systems Ollie and Mistral. Let's have a look at the `sst` forcing file options available in this file:

```
forcing_sources:
        # sst
        "amipsst":
                "${forcing_dir}/amip/${resolution}_amipsst_@YEAR@.nc":
                        from: 1870
                        to: 2016
        "pisst": "${forcing_dir}/${resolution}${ocean_resolution}_piControl-LR_sst_1880-
↪2379.ncy"
```

This means that from our runscript we will be able to select either `amipsst` or `pisst` as *sst* forcing files. If you define `scenario` in *ECHAM* be `PI-CTRL` the correct file source (`pisst`) is already selected for you. However, if you would like to select this file manually you can just simply add the following to your runscript:

```
forcing_files:
        sst: pisst
```

18.9.2 Modify the Source of a File

To change the path of the source for a given forcing or input file from your runscript:

1. Include the source path under a *key* inside `forcing_sources` or `input_sources` in your runscript:

   ```
   <forcing/input>_sources:
           <key_for_your_file>: <path_to_your_file>
   ```

   If the source is not a single file, but there is a file per year use the `@YEAR@` and `from: to:` functionality in the path to copy only the files corresponding to that run's year:

   ```
   <forcing/input>_sources:
           <key_for_your_source>: <firt_part_of_the_path>@YEAR@<second_part_of_the_
   ↪path>
   ```

(continues on next page)

(continued from previous page)

```
                from: <first_year>
                to: <last_year>
```

2. Make sure the *key* for your path is defined in one of the config files that you are using, inside of either `forcing_files` or `input_files`. If it is not defined anywhere you will have to include it in your runscript:

```
<forcing/input>_files:
        <key_for_your_file>: <key_for_your_source>
```

18.9.3 Copy the file in the work folder and/or rename it

To copy the files from the forcing/input folders into the work folder (`<base_dir>/<exp_id>/run_<DATE>/work`) or rename them:

1. Make sure your file and its source is defined somewhere (either in the config files or in your runscript) in `<forcing/input>_sources` and `<forcing/input>_files` (see subsections *Source Path Already Defined in a Config File* and *Modify the Source of a File*).
2. In your runscript, add the *key* to the file you want to **copy** with *value* the same as the *key*, inside <forcing/input>_in_work:

```
<forcing/input>_in_work:
        <key_for_your_file>: <key_for_your_file>
```

3. If you want to **rename** the file set the *value* to the desired name:

```
<forcing/input>_in_work:
        <key_for_your_file>: <key_for_your_file>
```

Example

ECHAM

In *ECHAM* the `sst` forcing file depends in the scenario defined by the user:

esm_tools/config/component/echam/echam.datasets.yaml

```
forcing_sources:
        # sst
        "amipsst":
                "${forcing_dir}/amip/${resolution}_amipsst_@YEAR@.nc":
                        from: 1870
                        to: 2016
        "pisst": "${forcing_dir}/${resolution}${ocean_resolution}_piControl-LR_sst_1880-
↪2379.nc"
```

esm_tools/config/component/echam/echam.yaml

```
choose_scenario:
        "PI-CTRL":
                forcing_files:
                        sst: pisst
                        [ ... ]
```

If `scenario: "PI-CTRL"` then the source selected will be `${forcing_dir}/${resolution}${ocean_resolution}_piControl-LR_sst_1880-2379.nc` and the name of the file copied to the experiment forcing folder will be `${resolution}${ocean_resolution}_piControl-LR_sst_1880-2379.nc`. However, *ECHAM* needs this file in the same folder as the binary (the `work` folder) under the name `unit.20`. To copy and rename this file into the `work` folder the following lines are used in the `echam.yaml` configuration file:

```
forcing_in_work:
        sst: "unit.20"
```

You can use the same syntax **inside your runscript** to copy into the `work` folder any forcing or input file, and rename it.

18.9.4 See also

- *What Is YAML?*
- *File Dictionaries*

18.10 Exclude a Forcing/Input File

Feature available since version: 4.2

To exclude one of the predefined forcing or input files from being copied to your experiment folder:

1. Find the *key* of the file to be excluded inside the config file, `<forcing/input>_files` *file dictionary*.
2. In your runscript, use the `remove_` functionality to exclude this *key* from the `<forcing/input>_files` *file dictionary*:

```
remove_<input/forcing>_files:
        - <key_of_the_file1>
        - <key_of_the_file2>
        - ...
```

18.10.1 Example

ECHAM

To exclude the `sst` forcing file from been copied to the experiment folder include the following lines in your runscript:

```
remove_forcing_files:
        - sst
```

18.10.2 See also

- *What Is YAML?*
- *Remove Elements from a List/Dictionary (remove_)*
- *File Dictionaries*

18.11 Using your own namelist

Feature available since version: 4.2

> **Warning:** This feature is only recommended if the number of changes that need to be applied to the default namelist is very large, otherwise we recommend to use the feature `namelist_changes` (see *Changing Namelist Entries from the Runscript*). You can check the default namelists here.

In your runscript, you can instruct *ESM-Tools* to substitute a given default namelist by a namelist of your choice.

1. Search for the `config_sources` variable inside the configuration file of the model you are trying to run, and then, identify the "key" containing the path to the default namelist.
2. In your runscript, indented in the corresponding model section, add an `add_config_sources` section, containing a variable whose "key" is the one of step 1, and the value is the path of the new namelist.
3. Bare in mind, that namelists are first loaded by *ESM-Tools*, and then modified by the default `namelist_changes` in the configuration files. If you want to ignore all those changes for the your new namelist you'll need to add `remove_namelist_changes: [<name_of_your_namelist>]`.

In dot notation both steps will look like: `<model_name>.<add_config_sources>.<key_of_the_namelist>: <path_of_your_namelist>` `<model_name>.<remove_namelis_changes>: [<name_of_your_namelist>]`

> **Warning:** Use step 3 at your own risk! Many of the model specific information and functionality is transferred to the model through `namelist_changes`, and therefore, we discourage you from using `remove_namelist_changes` unless you have a very deep understanding of the configuration file and the model. Following *Changing Namelist Entries from the Runscript* would be a safest solution.

18.11.1 Example

In this example we show how to use an *ECHAM* `namelist.echam` and a *FESOM* `namelist.ice` that are not the default ones and omit the `namelist_changes` present in `echam.yaml` and `fesom.yaml` configuration files.

ECHAM

FESOM

Following step 1, search for the `config_sources` dictionary inside the `echam.yaml`:

```
# Configuration Files:
config_sources:
        "namelist.echam": "${namelist_dir}/namelist.echam"
```

In this case the "key" is "namelist.echam" and the "value" is "${namelist_dir}/namelist.echam". Let's assume your namelist is in the directory /home/ollie/<usr>/my_namelists. Following step 2, you will need to include the following in your runscript:

```
echam:
        add_config_sources:
                "namelist.echam": /home/ollie/<usr>/my_namelists/namelist.echam
```

If you want to omit the namelist_changes in echam.yaml or any other configuration file that your model/couple setup is using, you'll need to add to your runscript remove_namelist_changes: [namelist.echam] (step 3):

```
echam:
        add_config_sources:
                "namelist.echam": /home/ollie/<usr>/my_namelists/namelist.echam

        remove_namelist_changes: [namelist.echam]
```

Warning: Many of the model specific information and functionality is transferred to the model through namelist_changes, and therefore, we discourage you from using this unless you have a very deep understanding of the echam.yaml file and the ECHAM model. For example, using remove_namelist_changes: [namelist.echam] will destroy the following lines in the echam.yaml:

```
choose_lresume:
        False:
                restart_in_modifications:
                        "[[streams-->STREAM]]":
                            - "vdate <--set_global_attr-- ${start_date!syear!smonth!
↪sday}"
                              # - fdate "<--set_dim--" ${year_before_date}
                              # - ndate "<--set_dim--" ${steps_in_year_before}
        True:
                # pseudo_start_date: $(( ${start_date} - ${time_step} ))
                add_namelist_changes:
                        namelist.echam:
                                runctl:
                                        dt_start: "remove_from_namelist"
```

This lines are relevant for correctly performing restarts, so if remove_namelist_changes is used, make sure to have the approrpiate commands on your runscript to remove dt_start from your namelist in case of a restart.

Following step 1, search for the config_sources dictionary inside the fesom.yaml:

```
config_sources:
        config:  "${namelist_dir}/namelist.config"
        forcing: "${namelist_dir}/namelist.forcing"
        ice:     "${namelist_dir}/namelist.ice"
        oce:     "${namelist_dir}/namelist.oce"
        diag:    "${namelist_dir}/namelist.diag"
```

In this case the "key" is ice and the "value" is ${namelist_dir}/namelist.ice. Let's assume your namelist is in the directory /home/ollie/<usr>/my_namelists. Following step 2, you will need to include the following in your runscript:

```
fesom:
        add_config_sources:
                ice: "/home/ollie/<usr>/my_namelists/namelist.ice"
```

If you want to omit the `namelist_changes` in `fesom.yaml` or any other configuration file that your model/couple setup is using, you'll need to add to your runscript `remove_namelist_changes: [namelist.ice]` (step 3):

```
fesom:
        add_config_sources:
                ice: "/home/ollie/<usr>/my_namelists/namelist.ice"

        remove_namelist_changes: [namelist.ice]
```

Warning: Many of the model specific information and functionality is transferred to the model through `namelist_changes`, and therefore, we discourage you from using this unless you have a very deep understanding of the `fesom.yaml` file and the FESOM model.

18.11.2 See also

- Default namelists on GitHub
- *Append to an Existing List (add_)*
- *Changing Namelists*
- *What Is YAML?*

18.12 How to branch-off FESOM from old spinup restart files

When you branch-off from very old FESOM ocean restart files, you may encounter the following runtime error:

```
read ocean restart file
Error:
NetCDF: Invalid dimension ID or name
```

This is because the naming of the NetCDF time dimension variable in the restart file has changed from `T` to `time` during the development of *FESOM* and the different *FESOM* versions. Therefore, recent versions of *FESOM* expect the name of the time dimension to be `time`.

In order to branch-off experiments from spinup restart files that use the old name for the time dimension, you need to rename this dimension before starting the branch-off experiment.

Warning: The following work around will change the restart file permanently. Make sure you do not apply this to the original file.

To rename a dimension variable of a NetCDF file, you can use `ncrename`:

```
ncrename -d T,time <copy_of_restart_spinup_file>.nc
```

where `T` is the old dimension and `time` is the new dimension.

18.12.1 See also

- cookbook:How to run a branch-off experiment

18.13 Recieve batch notifications via e-mail

It is possible to define the following variables in your runscript either in *computer* or in *general* (up to you which one) to recieve e-mail notifications about the status of your job:

```
general/computer:
    mail_type: <type_of_notification>
    mail_user: <your_e-mail>
```

Find more about the possible values of `mail_type` in SLURM documentation (https://slurm.schedmd.com/sbatch.html#OPT_mail-type)

18.14 AWI-ESM1/2 simulations with modified topography

18.14.1 Description

How to setup up a runscript to run an AWI-ESM1 / AWI-ESM2 simulation with modified topography.

Note: ECHAM6 becomes unstable if orography is suddenly significantly changed. Via the methodology provided below you can get your simulation into a stable state. The idea is that the perturbation of topography is not applied instantaneously but rather iteratively over one model year, thereby keeping the model in a stable numeric state.

18.14.2 Preparation steps

In order to set up a simulation with AWI-ESM1 or AWI-ESM2 (ECHAM6 as atmosphere model component) and modified topography, please do the following preparation steps that perform a bootstrapping of the atmosphere model for modified orography:

- Adapt all your model boundary conditions to the paleogeography that the simulation shall reflect, you may apply anomaly or absolut approaches.
- Adapt the configuration for the simulation to the settings as outlined in the example below.
- To avoid ECHAM6 becoming unstable if orography is suddenly significantly changed,
  - put `jansurf` file with the full paleoboundary condition, with all fields adapted, into the YAML setting `target_oro` under `echam -> add_input_sources`;
  - also prepare a version of the `jansurf` file that contains all paleo-boundary conditions, but has `GEOSP` and `OROXXX` data sets unchanged from modern; this adapted paleoboundary condition file is to be used in YAML setting `jansurf` under `echam -> add_input_sources`.
- If the change of orography shall be performed,
  - set the namelist switch `lupdate_orog` under `add_namelist_changes -> namelist.echam -> submodelctl` to `True`;

This switch will trigger ECHAM6 to read both `target_oro` and `jansurf`, using the latter as initial condition and then progressively adapting orography towards the condition in the former to reach a stable atmosphere state that is based on your modified orography.

- set the namelist switch `lupdate_orography` under `echam` to `True`;

  This triggers esm-tools to copy the additional boundary condition files to the work directory.

- run the model for one year;
- you may check that orography is adapted by searching for the following (and similar) strings in the log file of the awi-esm_compute of the ongoing run:

```
0:  PG: END of routine >>>read_target_orog_fields<<<
0:  PG: START of routine >>>calculate_adjustment_step_size<<<
0:  number_years =            1
0:  PG: There are this many timesteps:       70080
0:  PG: END of routine >>>calculate_adjustment_step_size<<<
0:  PG: Start of routine >>>update_orography<<<
0:  PG: Start of stepwise_update_orog_variable
0:  PG: END of stepwise_update_orog_variable
0:  PG: Start of stepwise_update_orog_variable
0:  PG: END of stepwise_update_orog_variable
0:  PG: Start of stepwise_update_orog_variable
0:  PG: END of stepwise_update_orog_variable
0:  PG: Start of stepwise_update_orog_variable
0:  PG: END of stepwise_update_orog_variable
0:  PG: Start of stepwise_update_orog_variable
0:  PG: END of stepwise_update_orog_variable
0:  PG: Start of stepwise_update_orog_variable
0:  PG: END of stepwise_update_orog_variable
0:  PG: Start of stepwise_update_orog_variable
0:  PG: END of stepwise_update_orog_variable
0:  PG: Start of stepwise_update_orog_variable
0:  PG: END of stepwise_update_orog_variable
0:  PG: The current difference from target in geosp is: 45556.2912600568
0:  PG: END of routine >>>update_orography<<<
```

- after this one year simulation with adaptation of orography to the target condition has finished, stop the simulation;

  Now you have produced a restart file that contains your target orography and a stable atmosphere state. This restart file may be used to initialize other simulations with your target orography or to simply continue your simulation. Regarding the latter:

- if the orography change has been sucessfully performed, then switch the setting `lupdate_orog` to `False` again, resubmit the simulation, and let the model equilibrate to the full paleo-boundary condition;

  You should not find any more log messages of the kind described above in your log files anymore.

18.14.3 Example

The following example YAML, that refers to a simulation that has been adapted to Pliocene geography, shows code snippets of additional changes to the configuration in order to activate a modified topography; these additional settings are shown in the context of other conventional YAML settings.

```
echam:

    input_sources:
        jansurf: "<path_to>/boundary_conditions/Pliocene_3Ma/echam6/T63CORE2_jan_surf.
↪Pliocene_3Ma.nc" # boundary condition adapted to paleogeography
        vgratclim: "<path_to>/boundary_conditions/Pliocene_3Ma/echam6/T63CORE2_VGRATCLIM.
↪Pliocene_3Ma.nc" # boundary condition adapted to paleogeography
        vltclim: "<path_to>/boundary_conditions/Pliocene_3Ma/echam6/T63CORE2_VLTCLIM.
↪Pliocene_3Ma.nc" # boundary condition adapted to paleogeography
# Code for bootstrapping:
    lupdate_orography: True #make esm-tools copy additional boundary conditions to the␣
↪work directory
    add_input_sources: #define both your target boundary condition and an initialization␣
↪version of it, where the latter contains modern orography
        jansurf: "<path_to>/boundary_conditions/Pliocene_3Ma/echam6/T63CORE2_jan_surf.
↪Pliocene_3Ma_modern_GEOSP.nc" # boundary condition adapted to paleogeography EXCEPT␣
↪FOR GEOSP AND OROXXX VARIABLES, these are as per standard modern setup
        target_oro: "<path_to>/boundary_conditions/Pliocene_3Ma/echam6/T63CORE2_jan_surf.
↪Pliocene_3Ma.nc" # boundary condition adapted to paleogeography, ALL FIELDS ADAPTED TO␣
↪THE DESIRED PALEOGEOGRAPHY THAT ECHAM6 SHOULD CONSIDER

    add_namelist_changes:
        namelist.echam:
            submodelctl:
                lupdate_orog: True #activates adaptation of orography from the␣
↪initialisation state to the target state

hdmodel:
    add_input_sources:
        hdpara: "<path_to>/boundary_conditions/Pliocene_3Ma/hd/hdpara.Pliocene_3Ma.nc"
↪#boundary condition adapted to paleogeography

jsbach:
    input_sources:
        jsbach_1850: "<path_to>/boundary_conditions/Pliocene_3Ma/jsbach/jsbach_T63CORE2_
↪11tiles_5layers_natural-veg.Pliocene_3Ma_semimoist.nc" #boundary condition adapted to␣
↪paleogeography

fesom:
    nx: 134288 #adapted to paleomesh
    mesh_dir: "<path_to>/boundary_conditions/Pliocene_3Ma/fesom2/midpli2/" #paleomesh
```

CHAPTER

NINETEEN

GLOSSARY

chunk
: A chunk is a set of *runs* that are grouped together for execution. This allows to group runs for offline coupling.

experiment
: The whole simulation and computations performed under the same *workflow*.

job
: A job is a unit of work inside a *workflow*. Sometimes also referred as a phase.

run
: A run the set of computations and *jobs* until a restart is performed, a cycle of within a *experiment*.

workflow
: A workflow is a series of *jobs* that are executed during a run and their dependencies.

CHAPTER

TWENTY

FREQUENTLY ASKED QUESTIONS

20.1 Installation

1. **Q**: My organization is not in the pull-down list I get when trying the Federated Login to gitlab.awi.de.

   A: Then maybe your institution just didn't join the DFN-AAI. You can check that at https://tools.aai.dfn.de/entities/.

2. **Q**: I am trying to use the Federated Login, and that seems to work fine. When I should be redirected to the gitlab server though, I get the error that my uid is missing.

   A: Even though your organization joined the DFN-AAI, gitlab.awi.de needs your organization to deliver information about your institutional e-mail address as part of the identity provided. Please contact the person responsible for shibboleth in your organization.

20.2 ESM Runscripts

1. **Q**: I get the error: `load_all_functions:  not found [No such file or directory]` when calling my runscript like this:

   ```
   $ ./my_run_script.sh -e some_expid
   ```

   A: You are trying to call your runscript the old-fashioned way that worked with the shell-script version, until revision 3. With the new python version, you get a new executable `esm_runscripts` that should be in your PATH already. Call your runscript like this:

   ```
   $ esm_runscripts my_run_script.sh -e some_expid
   ```

   All the command line options still apply. By the way, "load_all_function" doesn't hurt to have in the runscript, but can savely be removed.

2. **Q**: What should I put into the variable `FUNCTION_PATH` in my runscript, I can't find the folder `functions/all` it should point to.

   A: You can safely forget about `FUNCTION_PATH`, which was only needed in the shell script version until revision 3. Either ignore it, or better remove it from the runscript.

3. **Q**: When I try to branch-off from a spinup experiment using *FESOM*, I get the following runtime error:

   ```
   read ocean restart file
   Error:
   NetCDF: Invalid dimension ID or name
   ```

A: See How to branch-off FESOM from old spinup restart files.

20.3 ESM Master

1. **Q**: How can I define different environments for different models / different versions of the same model?

   A: You can add a choose-block in the models yaml-file inside the `computer` section (`esm_tools/configs/model_name.yaml`), e.g.:

```
<model>:
   version: 40r1
computer:
    choose_<model>.version:
        40r1:
            export_vars:
                MY_VAR: "something"
            module_actions:
                - load my_own_module
        43r3:
            export_vars:
               MY_VAR: "something_else"
```

2. **Q**: How can I add a new model, setup, and coupling strategy to the esm_master tool?

   A: Add your configuration in the file configs/esm_master/setups2models.yaml

20.4 Frequent Errors

1. **Q**: When I try to install *ESM-Tools* or use `esm_versions` I get the following error:

```
RuntimeError: Click will abort further execution because Python 3 was configured to␣
↪use ASCII as encoding for the environment. Consult https://click.palletsprojects.
↪com/en/7.x/python3/ for mitigation steps.
```

or something on the following lines:

```
 ERROR: Command errored out with exit status 1:
 command: /sw/rhel6-x64/conda/anaconda3-bleeding_edge/bin/python -c 'import sys,␣
↪setuptools, tokenize; sys.argv[0] = '"'"'/tmp/pip-install-0y687gmq/esm-master/setup.py'
↪"'"'; _file__='"'"'/tmp/pip-install-0y687gmq/esm-master/setup.py'"'"';
↪f=getattr(tokenize, '"'"'open'"'"', open)(__file__);code=f.read().replace('"'"'\r\n'"'"'
↪', '"'"'\n'"'"');f.close();exec(compile(code, _file__, '"'"'exec'"'"'))' egg_info --
↪egg-base /tmp/pip-install-0y687gmq/esm-master/pip-egg-info
 cwd: /tmp/pip-install-0y687gmq/esm-master/
 Complete output (7 lines):
 Traceback (most recent call last):
   File "<string>", line 1, in <module>
   File "/tmp/pip-install-0y687gmq/esm-master/setup.py", line 8, in <module>
     readme = readme_file.read()
   File "/sw/rhel6-x64/conda/anaconda3-bleeding_edge/lib/python3.6/encodings/ascii.py",␣
↪line 26, in decode
```

(continues on next page)

(continued from previous page)

```
    return codecs.ascii_decode(input, self.errors)[0]
UnicodeDecodeError: 'ascii' codec can't decode byte 0xf0 in position 1468: ordinal not␣
↪in range(128)
    ----------------------------------------
ERROR: Command errored out with exit status 1: python setup.py egg_info Check the logs␣
↪for full command output.

**A**: Some systems have ``C.UTF-8`` as locale default (i.e. ``$LC_ALL``, ``$LANG``).␣
↪This issue is solved by setting up the locales respectively to ``en_US.UTF-8`` and␣
↪``en_US.UTF-8``, either manually or adding them to the local bash configuration file␣
↪(i.e. ``~/.bash_profile``)::

    $ export LC_ALL=en_US.UTF-8
    $ export LANG=en_US.UTF-8
```

2. **Q**: How can I add a new model, setup, and coupling strategy to the esm_master tool?

   A: Add your configuration in the file `configs/esm_master/setups2models.yaml` (see contributing:Implementing a New Model and *Implement a New Coupled Setup*)

CHAPTER

TWENTYONE

PYTHON PACKAGES

The ESM-Tools are divided into a number of python packages / git repositories, both to ensure stability of the code as well as reusability:

21.1 esm_tools.git

The only repository to clone by hand by the user, `esm_tools.git` contains the subfolders

> **configs**: A collection of yaml configuration files, containing all the information needed by the python packages to work properly. This includes machine specific files (e.g. `machines/mistral.yaml`) , model specific files (e.g `fesom/fesom-2.0.yaml`), configurations for coupled setups (e.g. `foci/foci.yaml`), but also files with the information on how a certain software works (`batch_systems/slurm.yaml`), and finally, how the esm_tools themselves are supposed to work (e.g. `esm_master/esm_master.yaml`).

21.2 esm_master.git

This repository contains the python files that give the `esm_master` executable in the subfolder `esm_master`.

21.3 esm_runscripts.git

The python package of the `esm_runscripts` executable. The main routines can be found in `esm_runscripts/esm_sim_objects.py`.

21.4 esm_parser.git

In order to provide the additional functionality to the `yaml+` configuration files (like choose blocks, simple math operations, variable expansions etc.). `esm_parser` is an extension of the pyyaml package, it needs the `esm_calendar` package to run, but can otherwise easily be used to add `yaml+` configurations to any python software.

21.5 esm_calendar.git

CHAPTER

TWENTYTWO

ESM TOOLS CODE DOCUMENTATION

22.1 esm_archiving package

Top-level package for ESM Archiving.

esm_archiving.**archive_mistral**(*tfile*, *rtfile=None*)

Puts the `tfile` to the tape archive using `tape_command`

Parameters

- **tfile** (*str*) – The full path of the file to put to tape
- **rtfile** (*str*) – The filename on the remote tape server. Defaults to None, in which case a replacement is performed to keep as much of the filename the same as possible. Example: /work/ab0246/a270077/experiment.tgz –> /hpss/arch/ab0246/a270077/experiment.tgz

Return type
None

esm_archiving.**check_tar_lists**(*tar_lists*)

esm_archiving.**delete_original_data**(*tfile*, *force=False*)

Erases data which is found in the tar file.

Parameters

- **tfile** (*str*) – Path to the tarfille whose data should be erased.
- **force** (*bool*) – If False, asks the user if they really want to delete their files. Otherwise just does this silently. Default is `False`

Return type
None

esm_archiving.**determine_datestamp_location**(*files*)

Given a list of files; figures where the datestamp is by checking if it varies.

Parameters
files (*list*) – A list (longer than 1!) of files to check

Returns
A slice object giving the location of the datestamp

Return type
slice

Raises
DatestampLocationError : – Raised if there is more than one slice found where the numbers vary over different files -or- if the length of the file list is not longer than 1.

esm_archiving.**determine_potential_datestamp_locations**(*filepattern*)

For a filepattern, gives back index of potential date locations

Parameters
filepattern (*str*) – The filepattern to check.

Returns
A list of slice object which you can use to cut out dates from the filepattern

Return type
list

esm_archiving.**find_indices_of**(*char*, *in_string*)

Finds indicies of a specific character in a string

Parameters

- **char** (*str*) – The character to look for
- **in_string** (*str*) – The string to look in

Yields
int – Each round of the generator gives you the next index for the desired character.

esm_archiving.**get_files_for_date_range**(*filepattern*, *start_date*, *stop_date*, *frequency*, *date_format='%Y%m%d'*)

Creates a list of files for specified start/stop dates

Parameters

- **filepattern** (*str*) – A filepattern to replace dates in
- **start_date** (*str*) – The starting date, in a pandas-friendly date format
- **stop_date** (*str*) – Ending date, pandas friendly. Note that for end dates, you need to **add one month** to assure that you get the last step in your list!
- **frequency** (*str*) – Frequency of dates, pandas friendly
- **date_format** (*str*) – How dates should be formatted, defaults to %Y%m%d

Returns
A list of strings for the filepattern with correct date stamps.

Return type
list

Example

```
>>> filepattern =  "LGM_24hourly_PMIP4_echam6_BOT_mm_>>>DATE<<<.nc"
>>> LGM_files = get_files_for_date_range(filepattern, "1890-07", "1891-11", "1M",␣
↪date_format="%Y%m")
>>> LGM_files == [
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189007.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189008.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189009.nc",
```

(continues on next page)

(continued from previous page)

```
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189010.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189011.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189012.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189101.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189102.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189103.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189104.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189105.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189106.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189107.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189108.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189109.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189110.nc",
... ]
True
```

esm_archiving.**get_list_from_filepattern**(*filepattern*)

esm_archiving.**group_files**(*top*, *filetype*)

Generates quasi-regexes for a specific filetype, replacing all numbers with #.

Parameters

- **top** (*str*) – Where to start looking (this should normally be top of the experiment)
- **filetype** (*str*) – Which files to go through (e.g. outdata, restart, etc…)

Returns

A dictonary containing keys for each folder found in `filetype`, and values as lists of files with strings where numbers are replaced by #.

Return type

dict

esm_archiving.**group_indexes**(*index_list*)

Splits indexes into tuples of monotonically ascending values.

Parameters

list – The list to split up

Returns

A list of tuples, so that you can get only one group of ascending tuples.

Return type

list

Example

```
>>> indexes = [0, 1, 2, 3, 12, 13, 15, 16]
>>> group_indexes(indexes)
[(0, 1, 2, 3), (12, 13), (15, 16)]
```

esm_archiving.**log_tarfile_contents**(*tfile*)

Generates a log of the tarball contents

Parameters
tfile (*str*) – The path for the tar file to generate a log for

Return type
None

Warning: Note that for this function to work, you need to have write permission in the directory where the tarball is located. If not, this will probably raise an OSError. I can imagine giving the location of the log path as an argument; but would like to see if that is actually needed before implementing it…

esm_archiving.**pack_tarfile**(*flist*, *wdir*, *outname*)

Creates a compressed tarball (outname) with all files found in flist.

Parameters

- **flist** (*list*) – A list of files to include in this tarball
- **wdir** (*str*) – The directory to "change" to when packing up the tar file. This will (essentially) be used in the tar command as the -C option by stripping off the beginning of the flist
- **outname** (*str*) – The output file name

Returns
The output file name

Return type
str

esm_archiving.**purify_expid_in**(*model_files*, *expid*, *restore=False*)

Puts or restores >>>EXPID<<< marker in filepatterns

Parameters

- **model_files** (*dict*) – The model files for archiving
- **expid** (*str*) – The experiment ID to purify or restore
- **restore** (*bool*) – Set experiment ID back from the temporary marker

Returns
Dictionary containing keys for each model, values for file patterns

Return type
dict

esm_archiving.**sort_files_to_tarlists**(*model_files*, *start_date*, *end_date*, *config*)

esm_archiving.**split_list_due_to_size_limit**(*in_list*, *slimit*)

esm_archiving.**stamp_filepattern**(*filepattern*, *force_return=False*)

Transforms # in filepatterns to >>>DATE<<< and replaces other numbers back to original

Parameters

- **filepattern** (*str*) – Filepattern to get date stamps for
- **force_return** (*bool*) – Returns the list of filepatterns even if it is longer than 1.

Returns
New filepattern, with >>>DATE<<<

Return type
str

esm_archiving.**stamp_files**(*model_files*)

Given a sttandard file dictioanry (keys: model names, values: filepattern); figures out where the date probably is, and replaces the # sequence with a >>>DATE<<< stamp.

Parameters
model_files (*dict*) – Dictionary of keys (model names) where values are lists of files for each model.

Returns
As the input, but replaces the filepatterns with the >>>DATE<<< stamp.

Return type
dict

esm_archiving.**sum_tar_lists**(*tar_lists*)

Sums up the amount of space in the tar lists dictionary

Given `tar_lists`, which is generally a dicitonary consisting of keys (model names) and values (files to be tarred), figures out how much space the **raw, uncompressed** files would use. Generally the compressed tarball will take up less space.

Parameters
tar_lists (*dict*) – Dictionary of file lists to be summed up. Reports every sum as a value for the key of that particular list.

Returns
Keys are the same as in the input, values are the sums (in bytes) of all files present within the list.

Return type
dict

esm_archiving.**sum_tar_lists_human_readable**(*tar_lists*)

As `sum_tar_lists` but gives back strings with human-readable sizes.

22.1.1 Subpackages

esm_archiving.external package

Submodules

esm_archiving.external.pypftp module

class esm_archiving.external.pypftp.**Pftp**(*username=None*, *password=None*)

Bases: `object`

HOST = 'tape.dkrz.de'

PORT = 4021

close()

cwd(*path*)

change working directory

directories(*path=None*)
gather directories at the given path

static download(*source*, *destination*)
uses pftp binary for transfering the file

exists(*path*)
check if a path exists

files(*path=None*)
gather files at the given path

is_connected()
check if the connection is still active

isdir(*pathname*)
Returns true if pathname refers to an existing directory

isfile(*pathname*)
Returns true if pathname refers to an existing file

islink(*pathname*)

listdir(*path=None*)
list directory contents

listing(*path=None*)
list directory contents

listing2(*path=None*)
directory listing in long form. similar to "ls -l"

makedirs(*path*)
Recursively create dirs as required walking up to an existing parent dir

mkdir(*path*)

mlsd(*path*)

pwd()
present working directory

quit()

reconnect()
reconnects to the ftp server

remove(*filename*)

removedirs(*path*)

rename(*from_name*, *to_name*)

rmdir(*path*)
remove directory

size(*pathname*)
Returns size of path in bytes

stat(*pathname*)

Returns stat of the path

static upload(*source*, *destination*)

uses pftp binary for transfering the file

walk(*path=None*)

recursively walk the directory tree from the given path. Similar to os.walk

walk_for_directories(*path=None*)

recursively gather directories

walk_for_files(*path=None*)

recursively gather files

esm_archiving.external.pypftp.**download**(*source*, *destination*)

esm_archiving.external.pypftp.**upload**(*source*, *destination*)

22.1.2 Submodules

22.1.3 esm_archiving.cli module

After installation, you have a new command in your path:

```
esm_archive
```

Passing in the argument `--help` will show available subcommands:

```
Usage: esm_archive [OPTIONS] COMMAND [ARGS]...

  Console script for esm_archiving.

Options:
  --version             Show the version and exit.
  --write_local_config  Write a local configuration YAML file in the current
                        working directory
  --write_config        Write a global configuration YAML file in
                        ~/.config/esm_archiving/
  --help                Show this message and exit.

Commands:
  create
  upload
```

To use the tool, you can first `create` a tar archive and then use `upload` to put it onto the tape server.

Creating tarballs

Use `esm_archive create` to generate tar files from an experiment:

```
esm_archive create /path/to/top/of/experiment start_date end_date
```

The arguments `start_date` and `end_date` should take the form `YYYY-MM-DD`. A complete example would be:

```
esm_archive create /work/ab0246/a270077/from_ba0989/AWICM/LGM_6hours 1850-01-01 1851-01-
↪01
```

The archiving tool will automatically pack up all files it finds matching these dates in the `outdata` and `restart` directories and generate logs in the top of the experiment folder. Note that the final date (1851-01-1 in this example) is **not included**. During packing, you get a progress bar indicating when the tarball is finished.

Please be aware that are size limits in place on DKRZ's tape server. Any tar files **larger than 500 Gb will be trucated**. For more information, see: https://www.dkrz.de/up/systems/hpss/hpss

Uploading tarballs

A second command `esm_archive upload` allows you to put tarballs onto to tape server at DKRZ:

```
esm_archive upload /path/to/top/of/experiment start_date end_date
```

The signature is the same as for the `create` subcommand. Note that for this to work; you need to have a properly configured `.netrc` file in your home directory:

```
$ cat ~/.netrc
machine tape.dkrz.de login a270077 password OMITTED
```

This file needs to be readable/writable **only** for you, e.g. `chmod 600`. The archiving program will then be able to automatically log into the tape server and upload the tarballs. Again, more information about logging onto the tape server without password authentication can be found here: https://www.dkrz.de/up/help/faq/hpss/how-can-i-use-the-hpss-tape-archive-without-typing-my-password-every-time-e-g-in-scripts-or-jobs

22.1.4 esm_archiving.config module

When run from either the command line or in library mode (note **not** as an ESM Plugin), `esm_archiving` can be configured to how it looks for specific files. The configuration file is called `esm_archiving_config`, should be written in YAML, and have the following format:

```
echam:  # The model name
    archive: # archive seperator **required**
        # Frequency specification (how often
        # a datestamp is generated to look for)
        frequency: "1M"
        # Date format specification
        date_format: "%Y%m"
```

By default, `esm_archive` looks in the following locations:

1. Current working directory
2. **Any files in the XDG Standard:**
   https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html

If nothing is found, the program reverts to the hard-coded defaults, found in `esm_archiving/esm_archiving/config.py`

Note: In future, it might be changed that the program will look for an experiment specific configuration based upon the path it is given during the `create` or `upload` step.

Generating a configuration

You can use the command line switches `--write_local_config` and `--write_config` to generate configuration files either in the current working directory, or in the global directory for your user account defined by the XDG standard (typically ~/.config/esm_archiving):

```
$ esm_archive --write_local_config
Writing local (experiment) configuration...

$ esm_archive --write_config
Writing global (user) configuration...
```

esm_archiving.config.**load_config**()

Loads the configuration from one of the default configuration directories. If none can be found, returns the hard-coded default configuration.

Returns
A representation of the configuration used for archiving.

Return type
dict

esm_archiving.config.**write_config_yaml**(*path=None*)

22.1.5 esm_archiving.esm_archiving module

This is the `esm_archiving` module.

exception esm_archiving.esm_archiving.**DatestampLocationError**

Bases: `Exception`

esm_archiving.esm_archiving.**archive_mistral**(*tfile*, *rtfile=None*)

Puts the `tfile` to the tape archive using `tape_command`

Parameters

- **tfile** (*str*) – The full path of the file to put to tape
- **rtfile** (*str*) – The filename on the remote tape server. Defaults to None, in which case a replacement is performed to keep as much of the filename the same as possible. Example: /work/ab0246/a270077/experiment.tgz –> /hpss/arch/ab0246/a270077/experiment.tgz

Return type
None

esm_archiving.esm_archiving.**check_tar_lists**(*tar_lists*)

esm_archiving.esm_archiving.**delete_original_data**(*tfile*, *force=False*)

Erases data which is found in the tar file.

Parameters

- **tfile** (*str*) – Path to the tarfille whose data should be erased.
- **force** (*bool*) – If False, asks the user if they really want to delete their files. Otherwise just does this silently. Default is `False`

Return type
None

esm_archiving.esm_archiving.**determine_datestamp_location**(*files*)

Given a list of files; figures where the datestamp is by checking if it varies.

Parameters
files (*list*) – A list (longer than 1!) of files to check

Returns
A slice object giving the location of the datestamp

Return type
slice

Raises
DatestampLocationError : – Raised if there is more than one slice found where the numbers vary over different files -or- if the length of the file list is not longer than 1.

esm_archiving.esm_archiving.**determine_potential_datestamp_locations**(*filepattern*)

For a filepattern, gives back index of potential date locations

Parameters
filepattern (*str*) – The filepattern to check.

Returns
A list of slice object which you can use to cut out dates from the filepattern

Return type
list

esm_archiving.esm_archiving.**find_indices_of**(*char*, *in_string*)

Finds indicies of a specific character in a string

Parameters

- **char** (*str*) – The character to look for
- **in_string** (*str*) – The string to look in

Yields
int – Each round of the generator gives you the next index for the desired character.

esm_archiving.esm_archiving.**get_files_for_date_range**(*filepattern*, *start_date*, *stop_date*, *frequency*, *date_format='%Y%m%d'*)

Creates a list of files for specified start/stop dates

Parameters

- **filepattern** (*str*) – A filepattern to replace dates in
- **start_date** (*str*) – The starting date, in a pandas-friendly date format

- **stop_date** (*str*) – Ending date, pandas friendly. Note that for end dates, you need to **add one month** to assure that you get the last step in your list!
- **frequency** (*str*) – Frequency of dates, pandas friendly
- **date_format** (*str*) – How dates should be formatted, defaults to %Y%m%d

Returns
A list of strings for the filepattern with correct date stamps.

Return type
list

Example

```
>>> filepattern =  "LGM_24hourly_PMIP4_echam6_BOT_mm_>>>DATE<<<.nc"
>>> LGM_files = get_files_for_date_range(filepattern, "1890-07", "1891-11", "1M",␣
↪date_format="%Y%m")
>>> LGM_files == [
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189007.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189008.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189009.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189010.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189011.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189012.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189101.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189102.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189103.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189104.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189105.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189106.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189107.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189108.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189109.nc",
... "LGM_24hourly_PMIP4_echam6_BOT_mm_189110.nc",
... ]
True
```

esm_archiving.esm_archiving.**get_list_from_filepattern**(*filepattern*)

esm_archiving.esm_archiving.**group_files**(*top*, *filetype*)

Generates quasi-regexes for a specific filetype, replacing all numbers with #.

Parameters

- **top** (*str*) – Where to start looking (this should normally be top of the experiment)
- **filetype** (*str*) – Which files to go through (e.g. outdata, restart, etc…)

Returns
A dictonary containing keys for each folder found in `filetype`, and values as lists of files with strings where numbers are replaced by #.

Return type
dict

esm_archiving.esm_archiving.**group_indexes**(*index_list*)

Splits indexes into tuples of monotonically ascending values.

Parameters
list – The list to split up

Returns
A list of tuples, so that you can get only one group of ascending tuples.

Return type
list

Example

```
>>> indexes = [0, 1, 2, 3, 12, 13, 15, 16]
>>> group_indexes(indexes)
[(0, 1, 2, 3), (12, 13), (15, 16)]
```

esm_archiving.esm_archiving.**log_tarfile_contents**(*tfile*)

Generates a log of the tarball contents

Parameters
tfile (*str*) – The path for the tar file to generate a log for

Return type
None

Warning: Note that for this function to work, you need to have write permission in the directory where the tarball is located. If not, this will probably raise an OSError. I can imagine giving the location of the log path as an argument; but would like to see if that is actually needed before implementing it…

esm_archiving.esm_archiving.**pack_tarfile**(*flist*, *wdir*, *outname*)

Creates a compressed tarball (`outname`) with all files found in `flist`.

Parameters

- **flist** (*list*) – A list of files to include in this tarball
- **wdir** (*str*) – The directory to "change" to when packing up the tar file. This will (essentially) be used in the tar command as the -C option by stripping off the beginning of the flist
- **outname** (*str*) – The output file name

Returns
The output file name

Return type
str

esm_archiving.esm_archiving.**purify_expid_in**(*model_files*, *expid*, *restore=False*)

Puts or restores >>>EXPID<<< marker in filepatterns

Parameters

- **model_files** (*dict*) – The model files for archiving
- **expid** (*str*) – The experiment ID to purify or restore
- **restore** (*bool*) – Set experiment ID back from the temporary marker

Returns
Dictionary containing keys for each model, values for file patterns

Return type
dict

esm_archiving.esm_archiving.**query_yes_no**(*question*, *default='yes'*)

Ask a yes/no question via `input()` and return their answer.

"question" is a string that is presented to the user. "default" is the presumed answer if the user just hits <Enter>.

It must be "yes" (the default), "no" or None (meaning an answer is required of the user).

The "answer" return value is True for "yes" or False for "no".

Note: Shamelessly stolen from StackOverflow It's not hard to implement, but Paul is lazy...

Parameters

- **question** (*str*) – The question you'd like to ask the user
- **default** (*str*) – The presumed answer for `question`. Defaults to "yes".

Returns
True if the user said yes, False if the use said no.

Return type
bool

esm_archiving.esm_archiving.**run_command**(*command*)

Runs `command` and directly prints output to screen.

Parameters
command (*str*) – The command to run, with pipes, redirects, whatever

Returns
rc – The return code of the subprocess.

Return type
int

esm_archiving.esm_archiving.**sort_files_to_tarlists**(*model_files*, *start_date*, *end_date*, *config*)

esm_archiving.esm_archiving.**split_list_due_to_size_limit**(*in_list*, *slimit*)

esm_archiving.esm_archiving.**stamp_filepattern**(*filepattern*, *force_return=False*)

Transforms # in filepatterns to >>>DATE<<< and replaces other numbers back to original

Parameters

- **filepattern** (*str*) – Filepattern to get date stamps for
- **force_return** (*bool*) – Returns the list of filepatterns even if it is longer than 1.

Returns
New filepattern, with >>>DATE<<<

Return type
str

esm_archiving.esm_archiving.**stamp_files**(*model_files*)

Given a sttandard file dictioanry (keys: model names, values: filepattern); figures out where the date probably is, and replaces the # sequence with a >>>DATE<<< stamp.

Parameters
model_files (*dict*) – Dictionary of keys (model names) where values are lists of files for each model.

Returns
As the input, but replaces the filepatterns with the >>>DATE<<< stamp.

Return type
dict

esm_archiving.esm_archiving.**sum_tar_lists**(*tar_lists*)

Sums up the amount of space in the tar lists dictionary

Given `tar_lists`, which is generally a dicitonary consisting of keys (model names) and values (files to be tarred), figures out how much space the **raw, uncompressed** files would use. Generally the compressed tarball will take up less space.

Parameters
tar_lists (*dict*) – Dictionary of file lists to be summed up. Reports every sum as a value for the key of that particular list.

Returns
Keys are the same as in the input, values are the sums (in bytes) of all files present within the list.

Return type
dict

esm_archiving.esm_archiving.**sum_tar_lists_human_readable**(*tar_lists*)

As `sum_tar_lists` but gives back strings with human-readable sizes.

22.2 esm_calendar package

Top-level package for ESM Calendar.

22.2.1 Submodules

22.2.2 esm_calendar.esm_calendar module

Module Docstring.,..?

class esm_calendar.esm_calendar.**Calendar**(*calendar_type=1*)

Bases: `object`

Class to contain various types of calendars.

Parameters
calendar_type (*int*) – The type of calendar to use.

Supported calendar types: 0

no leap years

1
proleptic greogrian calendar (default)

n
equal months of `n` days

`timeunits`

A list of accepted time units.

Type
list of str

`monthnames`

A list of valid month names, using 3 letter English abbreviation.

Type
list of str

`isleapyear`(*year*)

Returns a boolean testing if the given year is a leapyear

`day_in_year`(*year*)

Returns the total number of days in a given year

`day_in_month`(*year*, *month*)

Returns the total number of days in a given month for a given year (considering leapyears)

`day_in_month`(*year*, *month*)

Finds the number of days in a given month

Parameters

- **year** (*int*) – The year to check
- **month** (*int or str*) – The month number or short name.

Returns
The number of days in this month, considering leapyears if needed.

Return type
int

Raises
TypeError – Raised when you give an incorrect type for month

`day_in_year`(*year*)

Finds total number of days in a year, considering leapyears if the calendar type allows for them.

Parameters
year (*int*) – The year to check

Returns
The total number of days for this specific calendar type

Return type
int

`isleapyear`(*year*)

Checks if a year is a leapyear

Parameters
year (*int*) – The year to check

Returns
True if the given year is a leapyear

Return type
bool

```
monthnames = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
'Nov', 'Dec']

timeunits = ['years', 'months', 'days', 'hours', 'minutes', 'seconds']
```

class esm_calendar.esm_calendar.**Date**(*indate*, *calendar=esm_calendar(calendar_type=1)*)

Bases: object

A class to contain dates, also compatiable with paleo (negative dates)

Parameters

- **indate** (*str*) – The date to use.

  See *pyesm.core.time_control.esm_calendar.Dateformat* for available formatters.

- **calendar** (*Calendar`, optional*) – The type of calendar to use. Defaults to a greogrian proleptic calendar if nothing is specified.

year

The year

Type
int

month

The month

Type
int

day

The day

Type
int

hour

The hour

Type
int

minute

The minute

Type
int

second

The second

Type
int

_calendar

The type of calendar to use

Type
Calendar`

add(*to_add*)

Adds another date to this one.

Parameters
to_add (*Date`*) – The other date to add to this one.

Returns
new_date – A new date object with the added dates

Return type
Date`

day_of_year()

Gets the day of the year, counting from Jan. 1

Returns
The day of the current year.

Return type
int

format(*form='SELF'*, *givenph=None*, *givenpm=None*, *givenps=None*)

Beautifully returns a Date object as a string.

Parameters

- **form** (*str or int*) – Logic taken from from MPI-Met
- **givenph** (*bool-ish*) – Print hours
- **givenpm** (*bool-ish*) – Print minutes
- **givenps** (*bool-ish*) – Print seconds

Note:

How to use the ``form`` argument
The following forms are accepted: + SELF: uses the format which was given when constructing the date + 0: A Date formatted as YYYY

In [5]: test.format(form=1) Out[5]: '1850-01-01_00:00:00'

In [6]: test.format(form=2) Out[6]: '1850-01-01T00:00:00'

In [7]: test.format(form=3) Out[7]: '1850-01-01 00:00:00'

In [8]: test.format(form=4) Out[8]: '1850 01 01 00 00 00'

In [9]: test.format(form=5) Out[9]: '01 Jan 1850 00:00:00'

In [10]: test.format(form=6) Out[10]: '18500101_00:00:00'

In [11]: test.format(form=7) Out[11]: '1850-01-01_000000'

In [12]: test.format(form=8) Out[12]: '18500101000000'

In [13]: test.format(form=9) Out[13]: '18500101_000000'

In [14]: test.format(form=10) Out[14]: '01/01/1850 00:00:00'

---

classmethod from_list(*_list*)

Creates a new Date from a list

Parameters
_list (*list of ints*) – A list of [year, month, day, hour, minute, second]

Returns
date – A new date of year month day, hour minute, second

Return type
Date`

classmethod fromlist(*_list*)

Creates a new Date from a list

Parameters
_list (*list of ints*) – A list of [year, month, day, hour, minute, second]

Returns
date – A new date of year month day, hour minute, second

Return type
Date`

makesense(*ndate*)

Puts overflowed time back into the correct unit.

When manipulating the date, it might be that you have "70 seconds", or something similar. Here, we put the overflowed time into the appropriate unit.

output(*form='SELF'*)

property sday

property sdoy

property shour

property sminute

property smonth

property ssecond

sub_date(*other*)

sub_tuple(*to_sub*)

Adds another date to from one.

Parameters
to_sub (*Date`*) – The other date to sub from this one.

Returns
new_date – A new date object with the subtracted dates

Return type
Date`

property syear

time_between(*date*, *outformat='seconds'*)

Computes the time between two dates

Parameters
date (*date`*) – The date to compare against.

Return type

??

class esm_calendar.esm_calendar.**Dateformat**(*form=1*, *printhours=True*, *printminutes=True*, *printseconds=True*)

Bases: object

datesep = ['', '-', '-', '-', ' ', ' ', '', '-', '', '', '/']

dtsep = ['_', '_', 'T', ' ', ' ', ' ', '_', '_', '', '_', ' ']

timesep = ['', ':', ':', ':', ' ', ':', ':', '', '', '', ':']

esm_calendar.esm_calendar.**date_range**(*start_date*, *stop_date*, *frequency*)

esm_calendar.esm_calendar.**find_remaining_hours**(*seconds*)

Finds the remaining full minutes of a given number of seconds

Parameters
seconds (*int*) – The number of seconds to allocate

Returns
The leftover seconds once new minutes have been filled.

Return type
int

esm_calendar.esm_calendar.**find_remaining_minutes**(*seconds*)

Finds the remaining full minutes of a given number of seconds

Parameters
seconds (*int*) – The number of seconds to allocate

Returns
The leftover seconds once new minutes have been filled.

Return type
int

22.3 esm_catalog package

22.3.1 Submodules

22.3.2 esm_catalog.cli module

22.3.3 esm_catalog.collection module

22.3.4 esm_catalog.context module

22.3.5 esm_catalog.item module

22.3.6 esm_catalog.registry module

22.4 esm_cleanup package

Cleanup tool for ESM-Tools simulations

22.4.1 Submodules

22.4.2 esm_cleanup.cli module

esm_cleanup.cli.**evaluate_arguments**()
The arg parser for interactive use

esm_cleanup.cli.**main**()

esm_cleanup.cli.**main_loop**(*folder*)

22.4.3 esm_cleanup.esm_cleanup module

esm_cleanup.esm_cleanup.**add_size_information**(*toplevel*, *item*)

esm_cleanup.esm_cleanup.**ask_for_action**(*folder*)

esm_cleanup.esm_cleanup.**assert_question**(*question*)

esm_cleanup.esm_cleanup.**check_if_folder_exists**(*folder*)

esm_cleanup.esm_cleanup.**dir_size**(*somepath*)

esm_cleanup.esm_cleanup.**file_size**(*somepath*)

esm_cleanup.esm_cleanup.**format_size**(*total_size*)

esm_cleanup.esm_cleanup.**inspect_size**(*thisfile*)

esm_cleanup.esm_cleanup.**is_experiment_folder**(*checkpath*)

esm_cleanup.esm_cleanup.**pick_experiment_folder**(*folder*)

esm_cleanup.esm_cleanup.**pick_subfolder**(*folder*)

esm_cleanup.esm_cleanup.**print_disclaimer**()

esm_cleanup.esm_cleanup.**print_folder_content**(*checkpath*, *saved_space*)

esm_cleanup.esm_cleanup.**read_in_yaml_file**(*filename*)

esm_cleanup.esm_cleanup.**remove_post_subfolders**(*folder*, *saved_space*)

esm_cleanup.esm_cleanup.**remove_run_subfolders**(*folder*, *saved_space*)

esm_cleanup.esm_cleanup.**remove_size_information**(*name_with_file*)

esm_cleanup.esm_cleanup.**remove_some_files**(*folder*, *saved_space*)

esm_cleanup.esm_cleanup.**remove_subfolder**(*folder*, *saved_space*)

22.5 esm_database package

Top-level package for ESM Database.

22.5.1 Submodules

22.5.2 esm_database.cli module

A small wrapper that combines the shell interface and the Python interface

esm_database.cli.**main**()

esm_database.cli.**parse_shargs**()
The arg parser for interactive use

22.5.3 esm_database.esm_database module

class esm_database.esm_database.**DisplayDatabase**(*tablename=None*)
Bases: object

ask_column()

ask_dataset()

decision_maker()

edit_dataset()

output_writer()

remove_datasets()

select_stuff()

22.5.4 esm_database.getch module

esm_database.getch.**get_one_of**(*testlist*)

22.5.5 esm_database.location_database module

class esm_database.location_database.**database_location**(***kwargs*)

Bases: Base

class_in

id

location

table_name

static topline()

esm_database.location_database.**register**(*table_name*, *given_location*, *class_in*)

22.6 esm_environment package

22.6.1 Submodules

22.6.2 esm_environment.esm_environment module

22.7 esm_master package

Top-level package for ESM Master.

22.7.1 Submodules

22.7.2 esm_master.cli module

22.7.3 esm_master.compile_info module

22.7.4 esm_master.database module

class esm_master.database.**installation**(***kwargs*)

Bases: Base

action

folder

id

static nicer_output(*run*)

setup_name

timestamp

static topline()

22.7.5 esm_master.database_actions module

esm_master.database_actions.**database_entry**(*config*, *action*, *setup_name*, *base_dir*)

22.7.6 esm_master.esm_master module

22.7.7 esm_master.general_stuff module

22.7.8 esm_master.software_package module

esm_master.software_package.**replace_var**(*var*, *tag*, *value*)

class esm_master.software_package.**software_package**(*raw*, *setup_info*, *vcs*, *general*, *no_infos=False*)

Bases: object

complete_targets(*setup_info*)

fill_in_infos(*setup_info*, *vcs*, *general*)

get_command_list(*setup_info*, *vcs*, *general*)

get_comp_type(*setup_info*)

get_coupling_changes(*setup_info*)

get_repo_info(*setup_info*, *vcs*)

get_subpackages(*setup_info*, *vcs*, *general*)

get_targets(*setup_info*, *vcs*)

output()

22.7.9 esm_master.task module

22.8 esm_motd package

22.8.1 Submodules

22.8.2 esm_motd.esm_motd module

22.9 esm_parser package

22.9.1 Submodules

22.9.2 esm_parser.dict_to_yaml module

esm_parser.dict_to_yaml.**add_eol_comments_with_provenance**(*commented_config*, *config*)

Add end-of-line comments to the `commented_config` with provenance information from the `config`.

Parameters

- **commented_config** (*dict*) – Dictionary with the config values and ruamel.yaml comments.
- **config** (*dict*) – Dictionary with the config values and provenance information.

esm_parser.dict_to_yaml.**delete_prev_objects**(*config*)

Delete key-values in the `config` which values correspond to `prev_` objects, for example, `prev_run` (contains values of the config of the previous run) and `prev_chunk_<model>` (that contains values of the config of a previous chunk in a offline coupled simulation). This deletion is necessary because otherwise the `finished_config.yaml` gets a lot of nested information from the previous runs that keeps growing each new run/chunk.

Parameters

config (*dict*) – Dictionary containing the simulation/compilation information

esm_parser.dict_to_yaml.**yaml_dump**(*config*, *config_file_path=None*)

Dump the config dictionary to a YAML file. The YAML file will contain comments with provenance information.

Parameters

- **config** (*dict*) – Dictionary containing the simulation/compilation information
- **config_file_path** (*str, optional*) – Path to the YAML file where the config will be saved. If not provided, the config will be printed to the standard output.

22.9.3 esm_parser.esm_parser module

22.9.4 esm_parser.provenance module

Provenance's dark magic. The basic idea is that one use the following to understand from which yaml file (line and column) a variable in `config` is coming from:

```
config["fesom"]["version"].provenance
```

And that will return a list of the provenance history of that variable, for example:

```
[{'category': 'components',
  'subcategory': 'fesom',
  'col': 10,
  'line': 6,
  'yaml_file': '/Users/mandresm/Codes/esm_tools/configs/components/fesom/fesom-2.0.yaml'}
↪,
 {'category': 'setups',
  'subcategory': 'awicm3',
  'col': 18,
  'extended_by': 'dict.__setitem__',
  'line': 321,
  'yaml_file': '/Users/mandresm/Codes/esm_tools/configs/setups/awicm3/awicm3.yaml'}]
```

The last element in the provenance list represents the provenance of the current value (last provenance).

This module contains: * The provenance class, to store the provenance of values with extended functionality * A wrapper factory to create classes and objects dynamically that subclass the value

types and append provenances to them (`WithProvenance` classes)

- Class attributes common to all `WithProvenance` classes
- **Classes for mappings with provenance (dictionaries and lists) to recursively put and**
  get provenance from nested values, and extend the standard mapping methods (`__setitem__`, `update`...)
- A decorator to keep provenance in `esm_parser`'s recursive functions
- A method to clean provenance recursively and get back the data without provenance

class esm_parser.provenance.**BoolWithProvenance**(*value*, *provenance=None*)

Bases: *ProvenanceClassForTheUnsubclassable*

Class for emulating `Bool` behaviour, but with Provenance.

Objects of this class reproduce the following `Bool` behaviours: * `isinstance(<obj>, bool)` returns `True` * `<True_obj> == True` returns `True` * `<True_obj> is True` returns `False`. This is not reproducing the behavior!

class esm_parser.provenance.**DictWithProvenance**(*dictionary*, *provenance*)

Bases: `dict`

A dictionary subclass that contains methods for: * recursively transforming leaf values into provenance (`put_provenance` and

`set_provenance`)

- recursively retrieving provenance from nested values
- **extending the `dict.__init__` method to recursively assign provenance to all**
  nested values
- **extending the `dict.__setitem__` method to keep a record of previous history**
  when adding new values to a given key
- **extending the `dict.update` method to keep a record of the previous history**
  when updating the dictionary

Use

Instance a new `DictWithProvenance` object:

```
dict_with_provenance = DictWithProvenance(<a_dictionary>, <my_provenance>)
```

Redefine the provenance of an existing `DictWithProvenance` with the same provenance for all its nested values:

```
dict_with_provenance.set_provenance(<new_provenance>)
```

Set the provenace of a specific leaf within a nested dictionary:

```
dict_with_provenance["key1"]["key1"].provenance = <new_provenance>
```

Get the `provenance` representation of the whole dictionary:

```
provenance_dict = dict_with_provenance.get_provenance()
```

extract_first_nested_values_provenance()

Recursively loops through the dictionary keys and returns the first provenance found in the nested values.

Returns
first_provenance – The first provenance found in the nested values

Return type
esm_parser.provenance.Provenance

get_provenance(*index=-1*)

Returns a `dictionary` containing the all the nested provenance information of the current `DictWithProvenance` with a structure and *keys* equivalent to the `self` dictionary, but with *values* of the *key* leaves those of the provenance.

Parameters
index (*int*) – Defines the element of the provenance history to be returned. The default is `-1`, meaning the last provenance (the one of the current value).

Returns
provenance_dict – A dictionary with a structure and *keys* equivalent to the `self` dictionary, but with *values* of the *key* leaves those of the provenance

Return type
dict

put_provenance(*provenance*)

Recursively transforms every value in `DictWithProvenance` into its corresponding WithProvenance object and appends its corresponding `provenance`. Each value has its corresponding provenance defined in the `provenance` dictionary, and this method just groups them together 1-to-1.

Parameters
provenance (*dict*) – The provenance that will be recursively assigned to all leaves of the dictionary tree. The provenance needs to be a `dict` with the same keys as `self` (same structure) so that it can successfully transfer each provenance value to its corresponding value on `self` (1-to-1 conrrespondance).

set_provenance(*provenance*, *update_method='extend'*)

Recursively transforms every value in `DictWithProvenance` into its corresponding WithProvenance object and appends the same `provenance` to it. Note that this method differs from `put_provenance` in that the same `provenance` value is applied to the different values of `self`.

Parameters

- **provenance** (*any*) – New *provenance value* to be set
- **update_method** (*str, optional*) – Method to use when updating provenance of existing values. Can be either `extend` to append the new provenance to the existing one, or `update` to update the last provenance entry with new values. Default is `extend`.

super_setitem(*key*, *val*)

A method to call the original `dict.__setitem__` method without provenance tracking. This is useful when you want to set a value without extending the provenance history, for example when you are setting a value that does not come from a yaml file and you do not want to keep the provenance history, or when resetting a value is blocked due to provenance conflicts.

update(*dictionary*, **args*, ***kwargs*)

Preserves the provenance history when using the `update` method

Parameters
dictionary (*dict, esm_parser.provenance.DictWithProvenance*) – Dictionary that will update `self`

yaml_dump(*config_file_path=None*)

Dump the config dictionary to a YAML file. The YAML file will contain comments with provenance information.

Parameters

- **config** (*dict*) – Dictionary containing the simulation/compilation information
- **config_file_path** (*str, optional*) – Path to the YAML file where the config will be saved. If not provided, the config will be printed to the standard output.

class esm_parser.provenance.**ListWithProvenance**(*mylist*, *provenance*)

Bases: `list`

A list subclass that contains methods for: * recursively transforming leaf values into provenance (`put_provenance` and

`set_provenance`)

- recursively retrieving provenance from nested values
- **extending the `list.__init__` method to recursively assign provenance to all** nested values
- **extending the `list.__setitem__` method to keep a record of previous history** when adding new values to a given key

Use

Instance a new `ListWithProvenance` object:

```
list_with_provenance = ListWithProvenance(<a_list>, <my_provenance>)
```

Redefine the provenance of an existing `ListWithProvenance` with the same provenance for all its nested values:

```
list_with_provenance.set_provenance(<new_provenance>)
```

Set the provenace of the element 0 of a list:

```
list_with_provenance[0].provenance = <new_provenance>
```

Get the provenance representation of the whole list:

```
provenance_list = list_with_provenance.get_provenance()
```

extract_first_nested_values_provenance()

Recursively loops through the list elements and returns the first provenance found in the nested values.

Returns
first_provenance – The first provenance found in the nested values

Return type
esm_parser.provenance.Provenance

get_provenance(*index=-1*)

Returns a `list` containing the all the nested provenance information of the current `ListWithProvenance` with a structure equivalent to the `self` list, but with list elements been provenance values.

Parameters
index (*int*) – Defines the element of the provenance history to be returned. The default is `-1`, meaning the last provenance (the one of the current value).

Returns
provenance_list – A list with a structure equivalent to that of the `self` list, but with the *values* of the provenance of each element

Return type
list

put_provenance(*provenance*)

Recursively transforms every value in `ListWithProvenance` into its corresponding WithProvenance object and appends its corresponding `provenance`. Each value has its corresponding provenance defined in the `provenance` list, and this method just groups them together 1-to-1.

Parameters
provenance (*list*) – The provenance that will be recursively assigned to all elements of the list. The provenance needs to be a `list` with the same number of elements as `self` (same structure) so that it can successfully transfer each provenance value to its corresponding value on `self` (1-to-1 conrrespondance).

set_provenance(*provenance*, *update_method='extend'*)

Recursively transforms every value in `ListWithProvenance` into its corresponding WithProvenance object and appends the same `provenance` to it. Note that this method differs from `put_provenance` in that the same `provenance` value is applied to the different values of `self`.

Parameters

- **provenance** (*any*) – New *provenance value* to be set
- **update_method** (*str, optional*) – Method to use when updating provenance of existing values. Can be either `extend` to append the new provenance to the existing one, or `update` to update the last provenance entry with new values. Default is `extend`.

super_setitem(*indx*, *val*)

A method to call the original `list.__setitem__` method without provenance tracking. This is useful when you want to set a value without extending the provenance history, for example when you are setting a value that does not come from a yaml file and you do not want to keep the provenance history, or when resetting a value is blocked due to provenance conflicts.

yaml_dump(*config_file_path=None*)

Dump the config dictionary to a YAML file. The YAML file will contain comments with provenance information.

Parameters

- **config** (*dict*) – Dictionary containing the simulation/compilation information
- **config_file_path** (*str, optional*) – Path to the YAML file where the config will be saved. If not provided, the config will be printed to the standard output.

class esm_parser.provenance.**NoneWithProvenance**(*value*, *provenance=None*)

Bases: *ProvenanceClassForTheUnsubclassable*

Class for emulating None behaviour, but with Provenance.

Objects of this class reproduce the following None behaviours: * isinstance(<obj>, None) returns True * <obj> == None returns True * <obj> is None returns False. This is not reproducing the behavior!

class esm_parser.provenance.**Provenance**(*provenance_data*)

Bases: list

A subclass of list in which each element represents the provenance of the value at a point in the key-value history. The whole point of this class is to have a list subclass that allows us to include information about which function is changing the list within each provenance element.

To assign the provenance to a value, instanciate it as an attribute of that value (i.e. self.provenance = Provenance(my_provenance)). To be used from the WithProvenance classes created by wrapper_with_provenance_factory.

The following class methods provide the extended functionality to lists: * self.append_last_step_modified_by: to duplicate the last element of the list

and add to it information about the function that is modifying the value

- **self.extend_and_modified_by: to extend a list while including in the** provenance the function which is responsible for extending it

add_modified_by(*provenance_step*, *func*, *modified_by='modified_by'*)

Adds a variable of name defined by modified_by to the given provenance step with value func. This variable is used to label provenance steps of the provenance history with functions that modified it.

Parameters

- **provenance_step** (*dict*) – Provenance entry of the current step
- **func** (*str*) – Function triggering this method
- **modified_by** (*str*) – Name of the key for the labelling the type of modification

Returns

provenance_step – Provenance entry of the current step with the modified_by item

Return type

dict

append_last_step_modified_by(*func*)

Copies the last element in the provenance history and adds the entry modify_by with value func to the copy.

Parameters

func (*str*) – Function that is modifying the variable

extend_and_modified_by(*additional_provenance*, *func*)

Extends the current provenance history with an `additional_provenance`. This happens when for example a variable comes originally from a file, but then the value is overwritten by another value that comes from a file higher in the hierarchy. This method keeps both histories, with the history of the second been on top of the first.

Parameters

- **additional_provenance** (*esm_parser.Provenance*) – Additional provenance history to be used for extending `self`
- **func** (*str*) – Function triggering this method

class esm_parser.provenance.**ProvenanceClassForTheUnsubclassable**(*value*, *provenance=None*)

Bases: `object`

A class to reproduce the methods of the unclassable `bool` and `NoneType` classes, needed to add the `provenance` attribute to those versions of the types.

This class stores 2 attributes, one is the `value` of the target object (a `bool` or a `NoneType`) and the other is the `provenance`.

property provenance

To be used as the `provenance` property in WithProvenance classes.

Returns
self._provenance – The provenance history stored in `self._provenance`

Return type
esm_parser.provenance.Provenance

esm_parser.provenance.**clean_provenance**(*data*, *nested=False*)

Returns the values of provenance mappings in their original classes (without the provenance). Recurs through mappings. Make sure you copy.deepcopy the data mapping before running this function if you don't want that your provenance information gets lost on the original `data` mapping.

Parameters
data (*any*) – Mapping or values with provenance.

Returns
value – Values in their original format, or lists and dictionaries containing provenance values.

Return type
any

esm_parser.provenance.**keep_provenance_in_recursive_function**(*func*)

Decorator for recursive functions in `esm_parser` to preserve provenance.

Parameters
func (*Callable*) – The function to decorate

esm_parser.provenance.**wrapper_with_provenance_factory**(*value*, *provenance=None*)

A function to subclass and instanciate all types of subclassable objects in the ESM-Tools `config` and add the `provenance` attribute to them. It also creates the `{type(value)}WithProvenance` classes globally on the fly depending on the `value`'s type, if it doesn't exist yet. For classes that are not subclassable (`Date`, `Bool` and `NoneType`) intanciates an object that mimics their behaviour but also contains the `provenance` attribute.

Objects of type `esm_calendar.esm_calendar.Date` are not subclassed (and the `provenance` attribute is simply added to them, because they fail to be subclassed with in the `DateWithProvenance` with the following error:

```
__new__ method giving error object.__new__() takes exactly one argument
(the type to instantiate)
```

Parameters

- **value** (*any*) – Value of the object to be subclassed and reinstanciated
- **provenance** (*any*) – The provenance information

Returns

- *{type(value)}WithProvenance, esm_calendar.esm_calendar.Date, BoolWithProvenance,*
- *NoneWithProvenance* – The new instance with the `provenance` attribute

`esm_parser.provenance.`**`wrapper_with_provenance_init`**`(`*self*, *value*, *provenance=None*`)`

To be used as the `__init__` method for WithProvenance classes. Adds the `provenance` value as an instance of `Provenance` to the `self._provenance` attribute, and stores the original `value` to the `self.value` attribute.

Parameters

- **value** (*any*) – Value of the object
- **provenance** (*any*) – The provenance information

`esm_parser.provenance.`**`wrapper_with_provenance_new`**`()`

To be used as the `__new__` method for WithProvenance classes. This is key for `copy.deepcopy`, without this `copy.deepcopy` breaks.

22.9.5 esm_parser.yaml_to_dict module

22.10 esm_plugin_manager package

22.10.1 Submodules

22.10.2 esm_plugin_manager.cli module

22.10.3 esm_plugin_manager.esm_plugin_manager module

22.11 esm_profile package

Top-level package for ESM Profile.

22.11.1 Submodules

22.11.2 esm_profile.esm_profile module

`esm_profile.esm_profile.`**`TIMING_INFO = {}`**

Global list of timed functions

esm_profile.esm_profile.**get_timing_info**()

Get timing information

Returns
list of timing information, as strings

Return type
list

esm_profile.esm_profile.**print_profile_summary**()

Prints the profile summary

esm_profile.esm_profile.**timing**(*f*, *task='other'*)

Decorator to time functions

Parameters
f (*callable*) – function to be timed

Returns
function with timing information

Return type
callable

22.12 esm_runscripts package

22.12.1 Submodules

22.12.2 esm_runscripts.assembler module

22.12.3 esm_runscripts.batch_system module

22.12.4 esm_runscripts.chunky_parts module

22.12.5 esm_runscripts.cli module

22.12.6 esm_runscripts.compute module

22.12.7 esm_runscripts.conda_env module

22.12.8 esm_runscripts.config_initialization module

22.12.9 esm_runscripts.coupler module

22.12.10 esm_runscripts.dask_cluster module

22.12.11 esm_runscripts.database module

22.12.12 esm_runscripts.database_actions module

22.12.13 esm_runscripts.dataprocess module

22.12.14 esm_runscripts.event_handlers module

22.12.15 esm_runscripts.filelists module

22.12.16 esm_runscripts.helpers module

22.12.17 esm_runscripts.inspect module

22.12.18 esm_runscripts.jinja module

22.12.19 esm_runscripts.last_minute module

22.12.20 esm_runscripts.logfiles module

22.12.21 esm_runscripts.methods module

22.12.22 esm_runscripts.mpirun module

22.12.23 esm_runscripts.namelists module

22.12.24 esm_runscripts.oasis module

22.12.25 esm_runscripts.observe module

22.12.26 esm_runscripts.pbs module

module, please refer to the handbook for user documentation as well as API documentation for the various sub-modules of the project.

Accessing Configuration

To access a particular configuration, you can use:

```
>>> from esm_tools import read_config_file
>>> ollie_config = read_config_file("machines/ollie")
```

Important note here is that the configuration file **has not yet been parsed**, so it's just the dictionary representation of the YAML.

esm_tools.**EDITABLE_INSTALL = False**

Shows if the installation is installed in editable mode or not.

Type
bool

esm_tools.**caller_wrapper**(*func*)

esm_tools.**copy_config_folder**(*dest_path*)

esm_tools.**copy_namelist_folder**(*dest_path*)

esm_tools.**copy_runscript_folder**(*dest_path*)

esm_tools.**get_config_as_str**(*config*)

esm_tools.**get_config_filepath**(*config=''*)

esm_tools.**get_coupling_filepath**(*coupling=''*)

esm_tools.**get_esm_tools_root_dir**()

Get the root directory of esm_tools installation.

This function finds the actual root directory of the esm_tools installation, which is useful for providing upgrade instructions or accessing the git repository.

For editable installs (pip install -e), this returns the git repository directory. For normal installs, this returns the site-packages location.

Returns
The root directory of esm_tools, or None if it cannot be determined.

Return type
pathlib.Path or None

Notes

This function is more reliable than _get_real_dir_from_pth_file("") because it: - Works regardless of virtual environment configuration - Doesn't depend on finding .egg-link files - Uses Python's standard __file__ attribute - Handles both editable and normal installations

esm_tools.**get_namelist_filepath**(*namelist=''*)

esm_tools.**get_runscript_filepath**(*runscript=''*)

esm_tools.**list_config_dir**(*dirpath*)

esm_tools.**read_config_file**(*config*)

Reads a configuration file, which should be seperated by "/". For example, "machines/ollie" will retrieve the (unparsed) configuration of the Ollie supercomputer.

Parameters
config (*str*) – Configuration to get, e.g. machines/ollie.yaml, or echam/echam. You may omit the ".yaml" ending if you want, it will be appended automatically if not already there.

Returns
A dictionary representation of the config.

Return type
dict

esm_tools.**read_namelist_file**(*nml*)

Reads a namelist file from a path, seperated by "/". Similar to `read_config_file`

Parameters
nml (*str*) – The namelist to load

Returns
A string of the namelist file

Return type
str

22.14.2 Submodules

22.14.3 esm_tools.cli module

22.14.4 esm_tools.error_handling module

esm_tools.error_handling.**user_error**(*error_type*, *error_text*, *exit_code=1*, *dsymbols=[''']*, *hints=[]*)

User-friendly error using `sys.exit()` instead of an `Exception`.

Parameters

- **error_type** (*str*) – Error type used for the error heading.
- **text** (*str*) – Text clarifying the error.
- **exit_code** (*int*) – The exit code to send back to the parent process (default to 1)

esm_tools.error_handling.**user_note**(*note_heading*, *note_text*, *color=\x1b[33m'*, *dsymbols=[''']*, *hints=[]*)

Notify the user about something. In the future this should also write in the log.

Parameters

- **note_heading** (*str*) – Note type used for the heading.
- **text** (*str*) – Text clarifying the note.

esm_tools.error_handling.**user_note_hints**(*note_text*, *hints*)

Add hints to the note text. The hints are added to the note text by replacing the placeholders "@HINT_#@" with the actual hints from the hints list.

Parameters

- **note_text** (*str*) – The note text with placeholders for the hints. The placeholders are in the form "@HINT_#@", where # is the index of the hint in the hints list.
- **hints** (*list*) – A list of hints to be added to the note text. Each hint is a dictionary with the following keys: - type: The type of the hint (e.g., "prov" for provenance) - text: The text of the hint. This text can contain a placeholder "@HINT@"

  which will be replaced with the actual hint corresponding to its index.

  - object: The object to which the hint applies

Returns
note_text – The note text with the placeholders replaced with the hints.

Return type
str

22.15 esm_utilities package

Top-level package for ESM Utilities.

22.15.1 Submodules

22.15.2 esm_utilities.cli module

Console script for esm_utilities.

22.15.3 esm_utilities.esm_utilities module

Main module.

22.15.4 esm_utilities.utils module

esm_utilities.utils.**check_valid_version**(*versionrange*, *version=''*)

Returns `True` if the `version` provided matches the condition of `versionrange`.

Parameters

- **version** (*str*) – String specifying the version number with the format `X.Y.Z`.
- **versionrange** (*str*) – Condition for the version range, expressed as a comparison operator followed by a version number in the format `X.Y.Z`.

Returns
True, False – `True` if the condition is met, `False` if not.

Return type
bool

esm_utilities.utils.**logfile_stats**(*logfile_to_read*)

CHAPTER

TWENTYTHREE

SUPPORTED MODELS

23.1 AMIP

23.2 DEBM

Institute	AWI
Description	dEBM is a surface melt scheme to couple ice and climate models in paleo applications.
Publications	Krebs-Kanzow, U., Gierz, P., and Lohmann, G., Brief communication: An Ice surface melt scheme including the diurnal cycle of solar radiation, The Cryosphere Discuss., accepted for publication
License	MIT

23.3 ECHAM

Institute	MPI-Met
Description	The ECHAM atmosphere model, major version 6
Authors	Bjorn Stevens (bjorn.stevens@mpimet.mpg.de) among others at MPI-Met
Publications	Atmosphericcomponent of the MPI-M earth system model: ECHAM6
License	Please make sure you have a license to use ECHAM. Otherwise downloading ECHAM will already fail. To use the repository on either gitlab.dkrz.de/modular_esm/echam6.git or gitlab.awi.de/paleodyn/models/echam6.git, please register for the MPI-ESM user forum at https://code.mpimet.mpg.de/projects/mpi-esm-license and send a screenshot of yourself logged in to the forum to either paul.gierz@awi.de, miguel.andres-martinez@awi.de, or nadine.wieters@awi.de. Note also that you can otherwise ignore the instructions on that page, just the registration and login screen shot is the relevant part for obtaining the license.

23.4 ESM_INTERFACE

Institute	Alfred Wegener Institute
Description	Coupling interface for a modular coupling approach of ESMs.
Authors	Nadine Wieters (nadine.wieters@awi.de)
License	None

23.5 FESOM

Institute	Alfred Wegener Institute for Polar and Marine Research (AWI)
Description	Multiresolution sea ice-ocean model that solves the equations of motion on unestructured meshes
Authors	Dmitry Sidorenko (Dmitry.Sidorenko@awi.de), Nikolay V. Koldunov (nikolay.koldunov@awi.de)
Publications	Danilov et al. 2004: A finite-element ocean model: principles and evaluation Wang et al. 2014: The Finite Element Sea Ice-Ocean Model (FESOM) v.1.4: formulation of an ocean general circulation model.
License	www.fesom.de

23.6 FESOM_MESH_PART

Description	The FESOM Mesh Partioner (METIS)

23.7 HDMODEL

23.8 ICON

Institute	MPI-Met
Description	The ICON atmosphere model, major version 2
Authors	Marco Giorgetta (marco.giorgetta@mpimet.mpg.de), Peter Korn, Christian Reick, Reinhard Budich
Publications	ICON-A, the Atmosphere Component of the ICON Earth System Model: I. Model Description
License	ICON is now open source, see https://www.icon-model.org

23.9 JSBACH

23.10 LPJ_GUESS

Institute	Lund Universitiy
Description	Dynamic vegetation model
Authors	Ben Smith
Publications	Smith, B., Prentice, I.C. & Sykes, M.T. 2001. Representation of vegetation dynamics in the modelling of terrestrial ecosystems: comparing two contrasting approaches within European climate space. Global Ecology & Biogeography 10: 621-637. Sitch, S., Smith, B., Prentice, I.C., Arneth, A., Bondeau, A., Cramer, W., Kaplan, J., Levis, S., Lucht, W., Sykes, M., Thonicke, K. & Venevsky, S. 2003. Evaluation of ecosystem dynamics, plant geography and terrestrial carbon cycling in the LPJ Dynamic Global Vegetation Model. Global Change Biology 9: 161-185. Martín Belda, D., Anthoni, P., Wårlind, D., Olin, S., Schurgers, G., Tang, J., Smith, B., & Arneth, A. 2022. LPJ-GUESS/LSMv1.0: A next generation Land Surface Model with high ecological realism. Geoscientific Model Development: Nord, J., Anthoni, P., Gregor, K., Gustafson, A., Hantson, S., Lindeskog, M., Meyer, B., Miller, P., Nieradzik, L., Olin, S., Papastefanou, P., Smith, B., Tang, J., Wårlind, D., & and past LPJ-GUESS contributors. (2021). LPJ-GUESS Release v4.1.1 model code (4.1.1). Zenodo.
License	Mozilla Public License Version 2.0

23.11 MEDUSA

Institute	University Liege
Description	MEDUSA is Model of Early Diagenesis in the Upper Sediment with Adatable complexity.
Authors	Guy Munhoven
Publications	Model of Early Diagenesis in the Upper Sediment with Adaptable complexity – MEDUSA (v. 2): a time-dependent biogeochemical sediment module for Earth system models, process analysis and teaching
License	Please make sure you have a licence to use MEDUSA. In case you are unsure, please contact Guy Munhoven (guy.munhoven@uliege.be)

23.12 MPIOM

Institute	MPI-Met
Description	The ocean-sea ice component of the MPI-ESM. MPIOM is a primitive equation model (C-Grid, z-coordinates, free surface) with the hydrostatic and Boussinesq assumptions made.
Authors	Till Maier-Reimer, Helmuth Haak, Johann Jungclaus
Publications	Characteristics of the ocean simulations in the Max Planck Institute Ocean Model (MPIOM) the ocean component of the MPI-Earth system model The Max-Planck-Institute global ocean/sea ice model with orthogonal curvilinear coordinates
License	Please make sure you have a licence to use MPIOM. In case you are unsure, please contact redmine…

23.13 NEMO

Organization	Nucleus for European Modelling of the Ocean
Institute	IPSL
Description	NEMO standing for Nucleus for European Modelling of the Ocean is a state-of-the-art modelling framework for research activities and forecasting services in ocean and climate sciences, developed in a sustainable way by a European consortium.
Authors	Gurvan Madec and NEMO System Team (nemo_st@locean-ipsl.umpc.fr)
Publications	NEMO ocean engine
License	Please make sure you have a license to use NEMO. In case you are unsure, please contact redmine…

23.14 NEMOBASEMODEL

23.15 OASIS3-MCT

Institute	CERFACS (Toulouse, France) and Centre National de la Recherche Scientifique (Paris, France)
Description	The OASIS coupler is a software allowing synchronized exchanges of coupling information between numerical codes representing different components of the climate system.
Author	OASIS team
Website	https://oasis.cerfacs.fr
Publications	Craig A., Valcke S., Coquart L., 2017: Development and performance of a new version of the OASIS coupler, OASIS3-MCT_3.0, Geoscientific Model Development, 10, pp. 3297-3308.
License	https://oasis.cerfacs.fr/en/oasis3-mct-copyright

23.16 OpenIFS

Institute	ECMWF
Description	OpenIFS provides research institutions with an easy-to-use version of the ECMWF IFS (Integrated Forecasting System).
Authors	Glenn Carver (openifs-support@ecmwf.int)
Website	https://www.ecmwf.int/en/research/projects/openifs
License	Please make sure you have a licence to use OpenIFS. In case you are unsure, please contact redmine…

23.17 PISM

Institute	UAF and PIK
Description	The Parallel Ice Sheet Model (PISM) is an open source, parallel, high-resolution ice sheet model.
Authors	Ed Bueler, Jed Brown, Anders Levermann, Ricarda Winkelmann and many more (uaf-pism@alaska.edu)
Publications	Shallow shelf approximation as a "sliding law" in a thermomechanically coupled ice sheet model The Potsdam parallel ice sheet model (PISM-PIK) - Part 1: Model description
License	GPL 3.0

23.18 PISM_NH

Institute	UAF and PIK
Description	The Parallel Ice Sheet Model (PISM) is an open source, parallel, high-resolution ice sheet model.
Authors	Ed Bueler, Jed Brown, Anders Levermann, Ricarda Winkelmann and many more (uaf-pism@alaska.edu)
Publications	Shallow shelf approximation as a "sliding law" in a thermomechanically coupled ice sheet model The Potsdam parallel ice sheet model (PISM-PIK) - Part 1: Model description
License	GPL 3.0

23.19 PISM_SH

Institute	UAF and PIK
Description	The Parallel Ice Sheet Model (PISM) is an open source, parallel, high-resolution ice sheet model.
Authors	Ed Bueler, Jed Brown, Anders Levermann, Ricarda Winkelmann and many more (uaf-pism@alaska.edu)
Publications	Shallow shelf approximation as a "sliding law" in a thermomechanically coupled ice sheet model The Potsdam parallel ice sheet model (PISM-PIK) - Part 1: Model description
License	GPL 3.0

23.20 RECOM

Institute	Alfred Wegener Institute for Polar and Marine Research (AWI)
Description	REcoM (Regulated Ecosystem Model) is an ecosystem and biogeochemistry model.
Authors	Judith Hauck, Ozgur Gurses
Publications	Seasonally different carbon flux changes in the Southern Ocean in response to the southern annular mode Arctic Ocean biogeochemistry in the high resolution FESOM 1.4-REcoM2 model
License	https://recom.readthedocs.io/en/latest/index.html

23.21 RNFMAP

23.22 SCOPE

Institute	Alfred Wegener Institute
Description	The Script-Based Coupler
Authors	Paul Gierz (pgierz@awi.de)

23.23 TUX

Institute	
Description	
Authors	
Publications	``_
License	GPL

23.24 VILMA

23.25 XIOS

Institute	IPSL and CEA
Description	A library dedicated to I/O management in climate codes.
Authors	Yann Meurdesoif (yann.meurdesoif@cea.fr)
Website	https://portal.enes.org/models/software-tools/xios
License	Please make sure you have a licence to use XIOS. In case you are unsure, please contact redmine…

23.26 YAC

Information	For more information about YAC please go to the webpage: https://dkrz-sw.gitlab-pages.dkrz.de/yac/index.html

23.27 YAXT

Information	For more information about YAXT please …
Description	yaxt
Authors	
License	

CHAPTER

TWENTYFOUR

TRANSITIONING FROM THE SHELL VERSION

24.1 ESM-Master

The Makefile based esm_master of the shell version has been replaced by a (python-based) executable called `esm_master` that should be in your PATH after installing the new tools. The command can be called from any place now, models will be installed in the current work folder. The old commands are replaced by new, but very similar calls:

```
OLD WAY:                          NEW WAY:
make                    -->       esm_master                    (to get the list of␣
↪available
                                                                 targets)
make get-fesom-1.4      -->       esm_master get-fesom-1.4      (download)
make conf-...           -->       esm_master conf-...           (configure)
make comp-...           -->       esm_master comp-...           (compile)
make clean-...          -->       esm_master clean-...          (clean)
```

Apart from that, the new esm_master offers certain new functionality:

```
esm_master fesom          (lists all available targets containing the string "fesom")
esm_master install-...    (shortcut for: get- , then conf- , then comp-)
esm_master recomp-...     (shortcut for: conf-, then clean-, then comp-)
esm_master log-...        (overview over last commits of the model, e.g. git log)
esm_master status-...     (changes in the model repository since last commit, e.g. git␣
↪status)
```

If the user wants to define own shortcut commands, that can be done by editing `esm_tools/configs/esm_master/esm_master.yaml`. New wrappers for the version control software can be e.g. added in `esm_tools/configs/vcs/git.yaml`. Adding commands in these configuration files is sufficient that they show up in the list of targets.

The details about models, setups, etc. are now to be found in `esm_tools/configs/esm_master/setups2models.yaml`. This file is a strucutred list instead of a barely readable, and rapidly growing, makefile. If you want to change details of your model, or add new components, this is where it should be put. Please refer to the chapter *ESM Master* for further details.

24.2 ESM-Environment

A visible tool, like esm-environment used to be, doesn't exist anymore. The information about the environment needed for compiling / running a model is contained:

- in the machine yaml file (e.g. `esm_tools/configs/machines/ollie.yaml`): This contains a default environment that we know works for a number of models / setups, but maybe not in an optimal way,
- in the model yaml file (e.g. `esm_tools/configs/fesom/fesom-2.0.yaml`): The model files can contain deviations from the default environment defined in the machine file, specified in the `computer` section of the configuration file.

Note: The previously used keywords `environment_changes`, `compiletime_environment_changes`, and `runtime_environment_changes` are now deprecated. Instead, use the `computer` section for all environment configurations.

Please note that even though there still is a python package called `esm_environment`, this is just the collection of python routines used to assemble the environment. It does not contain anything to be configured by the user.

24.3 ESM-Runscripts

One main thing that has changed for the runtime tool is the way it is evoked:

```
OLD WAY:                                NEW WAY:
./runscriptname -e experiment_id        esm_runscripts runscriptname -e experiment_id
```

Instead of calling your runscript directly, it is now interpreted and executed by the wrapper `esm_runscripts`, the second executable to be added to your PATH when installing the Tools. Internally, `esm_runscripts` reads in the script file line by line and converts it into a python dictionary. It is therefore also possible to write the "runscripts" in the form of a yaml file itself, which can be imported by python much easier. The user is invited to try the yaml-style runscripts, some example can be found in `esm_tools/runscripts`.

Some of the variables which had to be set in the script when using the shell version are now deprecated, these include:

- FUNCTION_PATH
- FPATH
- machine

Also the last two lines of the normel runscript for the shell version of the tools, `load_all_functions` and `general_do_it_all`, don't do anything anymore, and can be safely removed. They don't hurt though.

(. . . to be continued. . .)

24.4 Functions –> Configs + Python Packages

The shell functions, which used to be in `esm-runscripts/functions/all`, are gone. That was basically the whole point of re-coding the tools, to get rid of this mixture of model configuration, wild shell hacks, and in general lots of annoying problems. What used to be in the functions is now seperated into python code (which is actually doing things, but doesn't have any model-, setup- or machine specific information), and yaml configurations (which are basically structured lists of all the information we have, including mesh resolutions, scenario simulation forcings,...). Anything really that you could possibly know about running a simulation belongs into the yaml configs that you can now find in `esm_runscripts/configs`, while ESM-Tools functionality is coded in the python packages.

24.5 Namelists

No changes. Namelists can be found in `esm_tools/namelists`.

CHAPTER

TWENTYFIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

25.1 Types of Contributions

25.1.1 Report Bugs

Report bugs at https://github.com/esm-tools/esm_tools/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

25.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

25.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

25.1.4 Write Documentation

ESM Tools could always use more documentation, whether as part of the official ESM Tools docs, in docstrings, or even on the web in blog posts, articles, and such.

25.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/esm-tools/esm_tools/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

25.2 Get Started!

Ready to contribute? Here's how to set up *esm-tools* packages for local development (see *Python Packages* for a list of available packages). Note that the procedure of contributing to the *esm_tools* package (see *Contribution to esm_tools Package*) is different from the one to contribute to the other packages (*Contribution to Other Packages*).

25.2.1 Contribution to esm_tools Package

1. Fork the *esm_tools* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone https://github.com/esm-tools/esm_tools.git
   ```

   (or whatever subproject you want to contribute to).
3. By default, `git clone` will give you the release branch of the project. You might want to consider checking out the development branch, which might not always be as stable, but usually more up-to-date than the release branch:

   ```
   $ git checkout develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.
5. When you're done making changes, check that your changes pass flake8:

   ```
   $ flake8 esm_tools
   ```

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

25.2.2 Contribution to Other Packages

1. Follow steps 1-4 in *Contribution to esm_tools Package* for the desired package, cloning your fork locally with:

```
$ git clone https://github.com/esm-tools/<PACKAGE>.git
```

2. Proceed to do a development install of the package in the package's folder:

```
$ cd <package´s_folder>
$ pip install -e .
```

3. From now on when binaries are called, they will refer to the source code you are working on, located in your local package's folder. For example, if you are editing the package *esm_master* located in `~/esm_master` and you run `$ esm_master install-fesom-2.0` you'll be using the edited files in `~/esm_master` to install FESOM 2.0.
4. Follow steps 5-7 in *Contribution to esm_tools Package*.

Get Back to the Standard Distribution

Once finished with the contribution, you might want to get back to the standard non-editable mode version of the package in the `release` branch. To do that please follow these steps:

1. Uninstall all *ESM-Tools* packages (*Uninstall ESM-Tools*). This will not remove the folder where you installed the package in editable mode, just delete the links to that folder.
2. Navigate to the `esm_tools` folder and run the `./install.sh` script.
3. Check that your package is now installed in the folder `~/.local/lib/python3.<version>/site-packages/` .

Note: If the package is still shows the path to the editable-mode folder, try running `pip install --use-feature=in-tree-build .` from `esm_tools`.

25.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6, 3.7 and 3.8, and for PyPy. Check https://travis-ci.com/dbarbi/esm_tools/pull_requests and make sure that the tests pass for all supported Python versions.

25.4 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

CHAPTER

TWENTYSIX

HOW TO CITE THE SOFTWARE

When using the *ESM-Tools*, please cite the software by using the following DOI: https://zenodo.org/doi/10.5281/zenodo.3737927. This DOI represents all versions of the software, and will always points to the latest version on zenodo. If you navigate to that page you can also find the DOIs for each specific version, in the `Versions` panel.

Further information on available versions of the software can be found here.

CHAPTER

TWENTYSEVEN

CREDITS

27.1 Development Lead

- Dirk Barbi <dirk.barbi@awi.de>
- Paul Gierz <paul.gierz@awi.de>
- Nadine Wieters <nadine.wieters@awi.de>
- Miguel Andrés-Martínez <miguel.andres-martinez@awi.de>
- Deniz Ural <deniz.ural@awi.de>

27.2 Project Management

- Luisa Cristini <luisa.cristini@awi.de>

27.3 Contributors

- Sara Khosravi <sara.khosravi@awi.de>
- Fatemeh Chegini <fatemeh.chegini@mpimet.mpg.de>
- Joakim Kjellsson <jkjellsson@geomar.de>
- Sebastian Wahl <swahl@geomar.de>
- …

27.4 Beta Testers

- Tido Semmler <tido.semmler@awi.de>
- Christopher Danek <christopher.danek@awi.de>
- …

CHAPTER

TWENTYEIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

Y