

# **Make Experiments!**

Run-script generation for earth system models

Release 1.1.0dev

*Karl-Hermann Wieners  
Max-Planck-Institut für Meteorologie  
Hamburg*

# Table of Contents

1	Introduction.....	4
1.1	Example: ECHAM experiment setup.....	4
1.1.1	Experiments are defined by custom and default settings.....	4
1.1.2	Job templates are completed by settings to create scripts.....	5
1.1.3	Generating scripts only refers to the custom settings' file.....	7
2	Details on experiment definition.....	7
2.1	Design.....	8
2.2	Tools.....	8
2.3	Format of .config files.....	10
2.3.1	Variables.....	10
2.3.2	Sections.....	11
2.3.3	Special variables and sections.....	11
2.3.4	Variable interpolation.....	13
2.3.5	Evaluation of interpolation.....	14
2.3.6	Special expressions.....	15
2.4	Format of .tmpl files.....	16
2.4.1	Placeholders.....	16
2.4.2	Variables in sections.....	17
2.4.3	Expressions.....	17
2.4.4	Loops.....	19
2.4.5	Conditions.....	19
2.4.6	Comments.....	20
2.4.7	Block statements and block comments.....	20
2.5	Standard experiments.....	20
2.6	Standard options.....	21
2.6.1	Options set due to model configuration.....	21
2.7	Generating jobs.....	22
2.7.1	Changing the model job list.....	22
2.7.2	Pre-defined job variables.....	23
2.7.3	Overriding namelist settings in derived jobs.....	23
2.7.4	Native script variables.....	24
2.7.5	Initializing native script variables.....	25
2.7.6	Re-generation of scripts and backup.....	26
2.8	Standard environments.....	27
2.9	Defining namelists and other configuration files.....	27
2.9.1	Formatting the namelist information.....	28
2.9.2	Suppressing namelist groups or variables.....	28
2.9.3	Using the namelist text.....	29
2.9.4	Using native script variables in namelists.....	30
2.9.5	Non-namelist configuration files.....	31
2.10	Defining input files for an experiment.....	31
2.10.1	Overriding input files for certain jobs.....	32

# 1 Introduction

Running a numerical earth system model experiment requires a number of preparatory and processing steps like staging input data files, providing namelists and other configuration data, housekeeping duties like model-time management, post-processing and storing of output data. These steps are usually put into scripts or another kind of job description that is finally executed on some high-performance computing system.

The Make Experiments! (mkexp) toolbox provides a generic interface to setting up such an experiment. At the heart of this lies the so-called .config file. This is a simple text file that contains the model settings for your experiment in a way that is largely independent of the job description that is later used to run it.

To make this an easy task, MPI-M's models are delivered with a number of standard experiment types. Your own .config file will choose one of these, thus inheriting their settings for use in your experiment. Typically, it will also override or amend these settings for the purpose of your experiment.

While the .config file contains all necessary settings, there is much more to the actual job description. Therefore, the standard experiment types also provide templates (.tmpl files) for the jobs to run, that are then converted to the final job step descriptions, using the .config file settings. Besides, the .config file contains a 'jobs' section, where the job control flow and parameters of the job description itself may be adjusted, e.g. requiring more resources or disabling certain job steps.

## 1.1 Example: ECHAM experiment setup

To illustrate the way *mkexp* works, let us look at the way that experiments with ECHAM, MPI-M's atmospheric circulation model, are set up.

### 1.1.1 Experiments are defined by custom and default settings

ECHAM comes with five pre-defined experiment types, amip-LR, amip-MR, amip-HR, sstClim-LR and sstClim-MR. To set up an experiment based on one of these, like amip-LR, all you have to do is create your own experiment configuration file within ECHAM's run subdirectory, e.g. 'joe1234.config', setting amip-LR as experiment type and providing an experiment description with the header comment:

```
# Standard AMIP experiment as baseline for further experiments (LR)

EXP_TYPE = amip-LR
```

For each experiment type, you will find a .config file in the run/standard\_experiments subdirectory. For instance, 'amip-LR.config' includes these settings

```
# Default definitions for amip-LR experiments

RES = 63

[namelists]

  [[namelist.echam]]
    [[[runctl]]]
      lamip = true

  [[namelist.jsbach]]
    [[[jsbach_ctl]]]
      use_dynveg = false
```

As you can see, settings are simple name = value pairs that may be organized in sections. Sections are defined by a bracketed section name. They may contain subsections, where the number of brackets defines the hierarchy depth. The pre-defined sections [namelists] and [jobs] are used for special purposes within *mkexp*.

Some settings apply to all ECHAM experiment types. They go into a file named 'DEFAULT.config'. *mkexp* will always read this file first, before applying any settings from a specific experiment type like 'amip-LR.config'. Such settings might be default paths for input data, as in

```
# Default definitions for all ECHAM experiments

ATMO_INPUT_ROOT = /pool/data/ECHAM6/input/r0004
LAND_INPUT_ROOT = /pool/data/JSBACH/input/r0004
```

The final configuration is then merged from all of these three sources, where the experiment configuration may override or amend settings from the experiment type, and the type config may in turn change the model defaults.

### 1.1.2 Job templates are completed by settings to create scripts

Now the experiment configuration is finalized, the settings are used to fill in a kind of job description forms, so called *templates*, from which eventually the actual job description files are created. As the structure of jobs is largely independent of the model resolution, all amip- experiment types share the same set of template files. One of them is 'amip.run.tmpl', also within the run/standard\_experiments subdirectory. The excerpt below shows that this looks mostly like a shell script, but contains placeholders that are filled using the .config file information:

```

#!/bin/bash

# Job file to run ECHAM 6

EXP=%{EXP_ID} # experiment identifier

RES=%{RES} # experiment truncation

# absolute paths to directories with initial data:
ATMO_DATA=%{ATMO_INPUT_ROOT}
ATMO_MAP_DATA=$ATMO_DATA/T${RES}

# absolute path to directory with initial data for JSBACH:
LAND_MAP_DATA=%{LAND_INPUT_ROOT}/T${RES}

#
# ECHAM6 namelist
#
cat > namelist.echam << EOF
%{NAMELIST_ECHAM}
EOF

#
# JSBACH namelist
#
cat > namelist.jsbach << EOF
%{NAMELIST_JSBACH}
EOF

```

These '%{...}' constructs correspond to the configuration settings shown in the previous section. The value for the setting given by the variable name between '%{' and '}' is pasted into the template, replacing the placeholder. The special names 'NAMELIST\_ECHAM' and 'NAMELIST\_JSBACH' contain the contents of the 'namelists' subsections, with each setting taken to be a Fortran namelist setting, and formatted accordingly. 'EXP\_ID' is taken to be the base filename of the experiment's .config file. The result of this operation is then written to the final job script, in our case 'joe1234.run':

```

#!/bin/bash

# Job file to run ECHAM 6

EXP=joe1234 # experiment identifier

RES=63 # experiment truncation

# absolute paths to directories with initial data:
ATMO_DATA=/pool/data/ECHAM6/input/r0004
ATMO_MAP_DATA=$ATMO_DATA/T${RES}

# absolute path to directory with initial data for JSBACH:
LAND_MAP_DATA=/pool/data/JSBACH/input/r0004/T${RES}

#
# ECHAM6 namelist
#
cat > namelist.echam << EOF
&runctl
    lamip = .true.
/
EOF

#
# JSBACH namelist
#
cat > namelist.jsbach << EOF
&jsbach_ctl
    use_dynveg = .false.
/
EOF

```

### 1.1.3 Generating scripts only refers to the custom settings' file

So, as soon as you have set up 'joe1234.config', you may create the job scripts that are needed to run your experiment. Still within ECHAM's 'run' subdirectory, type the following into your terminal:

```
../util/mkexp/mkexp joe1234.config
```

This will read all configuration information and create all job scripts that are defined within your experiment's setup, using their respective templates as shown in the previous section. *mkexp* will put these scripts in a common directory defined by the .config variable 'SCRIPT\_DIR', and print the name of this directory on your terminal.

## 2 Details on experiment definition

With the introductory example of the previous section in mind, this section will give some more detailed information on specific aspects of *mkexp*.

## 2.1 Design

Experiment definition with *mkexp* is organized in three levels.

The first level is the *mkexp* toolbox. It provides the front end to create an executable job description from a generic experiment configuration, but does not contain any model specific information. Instead a basic set of conventions is defined that should be applicable to a very large range of model systems. When this document uses the term *mkexp*, it refers to this system level.

At a second level, a model needs to provide a number of files containing the information needed by *mkexp*: the required job steps and their interaction, the basic contents of job scripts, model specific information, and building blocks that may be combined to define a specific experiment. This is called the *model setup*. The files must maintain the naming conventions prescribed by *mkexp*.

Finally, the third level is the actual experiment definition. Here the user decides which of the building blocks from the previous level are needed, and defines experiment specific settings that override or amend the information from the model setup. It is also essential to supply an experiment description and a – possibly unique – experiment identifier. All this is called the *user setup*.

All levels should make a clear distinction between the .config files, containing the experiment's configuration information, and the .tmpl files, containing the actual job description and job control syntax.

## 2.2 Tools

The *mkexp* package provides a number of tools for working with script configurations and setups.

```
mkexp [-m] [-g] file.config [name=value ...]
```

This is the main tool for generating an experiment setup. It takes the given user setup and the model setup that is referenced by the user setup to generate the job description files or scripts that are required to run a model experiment as specified in *file.config*.

When running, *mkexp* creates three directories, one each for the job scripts, run-time data, and output data, as defined by the setup. The names of these are printed, plus warnings if they already exist.

*mkexp* allows to override or amend the .config file settings on the command line by defining or re-defining a variable *name* set to *value*. Section variables are referenced as *sectionname.variablename*. Any periods in the variable name have to be duplicated, e.g. to set '.remove' in section 'jobs' to 'post', use 'jobs...remove=post'. Note that three periods will always be read as '.' followed by '.', thus it is not possible to use variable names that *end* in a period.

When given the '-m' or '--no-make-dirs' option, only the script directory is created while creation of the run-time and output directories is skipped.

With '-g' or '--getexp', instead of a .config file, *mkexp* expects a dump generated by *getexp -vv* (see below). The experiment setup is regenerated from this dump, overriding any model setup.

```
getexp [-v ...] [-R] [-k key] file.config [name=value ...]
```

*getexp* reads the experiment setup the same way as *mkexp*, but does not generate job scripts. Instead it prints the experiment name and directories to be generated in a shell-readable form. It is intended for debugging or passing setup information to utility scripts.

When given the '-v' or '--verbose' option, all *global* configuration variables and their values are printed in alphabetical order. When given twice, the whole configuration is dumped to the screen. Save this to a file for use with *mkexp -g*. When given the '-R' or '--readme' option, the header comment text is printed. When given the '-k' or '--key' option, only the configured value for *key* is printed. Section variables may be referenced as described above for *name=value*. This option may be used more than once to print additional values.

```
diffexp file1.config file2.config
```

For an easy comparison of the whole set of generated scripts for two different experiments, this tool takes the directories defined in each configuration, locates the job scripts corresponding to each other (e.g. exp0001.run and exp0002.run), equalizes all occurrences of the experiment name in the scripts and then uses the diff tool to show differences. The environment variable 'DIFF' may be set to an alternative tool to be called instead.

```
rmexp file.config [name=value ...]
```

This allows interactive removal for all data of an experiment without having to deal with path names, as these are read from the configuration.

```
cpexp [-n] file.config new_name [name=value ...]
```

Replicates all data of an experiment to a new experiment name; also updates text files by rewriting references to the old name. With '-n', shows what would be done instead of actually doing it

```
duexp file.config [name=value ...]
```

Shows disk usage for all data that has been created by an experiment.

```
upexp file.config [name=value ...]
```

Update generated scripts for the given experiment with the same *mkexp* version, environment and command line, as saved in the corresponding 'update' script.

```
editexp [file]
```

Reads the update script *file* ('update' by default) and launches a program to edit the corresponding config file. The program is taken from the environment variables 'VISUAL' or 'EDITOR' if defined, otherwise *vi* is launched.

```
getconfig [file]
```

Documentation tool for experiments that were created using command line assignments. Reads the update script *file* ('update' by default) and prints the corresponding config file with command line settings from the update script



included.

```
setconfig [-d key] [-H text] [-a file.config] [file.config [name=value ...]]
```

Filter tool to alter configuration files via command line. Reads *file.config* (standard input by default or if *file.config* = '-') and prints the filtered configuration to standard output. Add or alter variables by *name=value* as described before. With '-d' or '--delete', the variable *key* is removed from the configuration. With '-H' or '--header', *text* is appended to the configuration's header comment. For files given with '-a' or '--add', all settings are merged with *file.config*.

```
namelist2config [-d [-c] [-v]]
```

Tries to extract namelist settings from shell scripts or log files and converts them to the .config format. By default, comments are ignored and names of namelist groups and variables sorted to allow easier comparisons. With '-d', namelists are printed directly, in original order. In this mode, '-c' enables comments, '-v' will output non-namelist lines from input files as comments, prefixed with '###'

## 2.3 Format of .config files

The .config files are simple text files containing a dictionary of variables with their respective values. They may be structured using sections and comments. For reading these files, *mkexp* uses the *configobj* Python library. All settings found in the .config files are handled as Python variables internally.

### 2.3.1 Variables

A configuration variable is set by simply assigning a text value to a name, as in

```
NAME = Joe User
```

Note that spaces before and after the 'equals' sign are always ignored. The value starts with the first non-space character. Spaces and additional equals after this are part of the value. In the case above, the variable NAME is set to 'Joe User'. To include leading spaces, you may enclose the actual value in single or double quote characters as in

```
SEPARATOR = '          '
```

Comma separated values are taken to be a list of string values. Thus

```
PATH = /bin, /usr/bin, /usr/local/bin
```

will set PATH to ['/bin', '/usr/bin', '/usr/local/bin'].

### 2.3.2 Sections

Variable assignments may be contained in *sections*. They group a set of variables that may be treated in a way different from the global variables. Sections are created by a section name on a line by itself, enclosed by brackets. Any variables defined later in the

.config file belong to this section:

```
[section1]
  description = This is the first section
```

will be stored as a dictionary section1 with section1['description'] set to 'This is the first section'.

Sections may be nested to arbitrary depth by incrementing the number of bracket pairs as in

```
[section1]
  description = This is the first section
  [[subsection1a]]
    description = This is the first sub section of the first section
  [[subsection1b]]
    description = This is the second sub section of the first section
[section2]
  description = This is the second section
```

A section is closed by the beginning of a new section of the same level, by a section of lower nesting depth, or the end of the .config file. Thus, section1 will contain 'description' and two dictionaries 'subsection1' and 'subsection2', each of those containing their own 'description'. 'section2' then is a top-level dictionary, again with its own 'description' variable.

Note that indentation may be used to make the file more legible but is completely ignored when the file is loaded. The number of brackets is the only way to define the level of a section. This means that all variables in a section must be defined before any subsections. Otherwise, the variable would belong to the respective subsection.

Fortran scholars will also want to note that names are case-sensitive, i.e. the variable 'NAME' is quite different from 'name'. Usually, setups use upper-case names for global variables and lower-case names for sections and their variables.

### 2.3.3 Special variables and sections

There are a number of special variables that influence the way *mkexp* works. They must be present in one of the .config files, unless noted otherwise below. They are listed here for a first overview. Their exact meaning is explained in more detail in the upcoming sections.

The first set of variables is usually defined in the model setup:

#### SCRIPT\_DIR

Directory where the generated job descriptions are stored. This directory and its parents are created by *mkexp* if they do not exist.

#### WORK\_DIR

Directory where the experiment is run. The jobs will use this for providing input data and configuration files needed for model execution. This directory and its parents are created if they do not exist.

## DATA\_DIR

Directory for storing output data. When a model run finishes, output will be stored there for further processing. Will also be created when non-existent.

## VERSION\_

Each .config file in the model setup should set this variable to a suitable value, e.g. version control information. The values are collected in a variable 'VERSIONS\_' which is usually written to the resulting job descriptions.

## SETUP\_OPTIONS (*optional*)

Subset of the model's standard options that should be applied to all experiments using the same model version.

There is a second set of variables that belongs in the user setup:

## EXP\_TYPE

Selects one of the standard experiments that are pre-defined in the model setup as basis of the current experiment definition.

## ENVIRONMENT

Selects one of the standard host environments that are available for the model.

## EXP\_OPTIONS (*optional*)

Subset of the model's standard options that should be applied to the current experiment definition.

## EXP\_ID (*optional*)

Name of the experiment to be created. If not set, this will be set to the base name of the user's .config file, e.g. 'joe1234' in the introductory example. All job description files will carry this as the first part of their name. For almost all model setups, this will be used in the definitions of SCRIPT\_DIR, WORK\_DIR, and DATA\_DIR.

## EXP\_DESCRIPTION (*optional*)

Extensive description of the experiment to be created. If not set, this will contain all text in the header comment of the user's .config file. The leading comment characters, as well as leading and trailing empty lines or comment boilerplate are removed. Note that both header comment and EXP\_DESCRIPTION may reference any other global variable defined in the experiment configuration (see section 2.3.4).

The contents of this variable is written to a 'README' file in SCRIPT\_DIR.

Another set of variables is automatically added to the job specific experiment configuration. These are considered read-only and may not be altered.

## JOB

A dictionary of system settings pertaining to the current job.

## VARIABLES\_

List of all names that were recognized as native variables of the current job. May be used to maintain a variable definition list in the generated script.

## mkexp\_input

Descriptive string for script headers. It is set to 'Generated by ... mkexp ...' where the ellipses are filled with version information.

## VERSIONS\_

List of all 'VERSION\_' strings that were found in the different .config files.

These special sections are usually pre-defined in the model setup, but are commonly

altered by the user.

[jobs]

This section defines the job description set needed for an experiment. It also provides job specific settings. Details are given in section 2.7, 'Generating jobs'.

[namelists]

Information that is contained in model configuration or namelist files is set in this section. For further details see section 2.9, 'Defining namelists and other configuration files'.

[files]

All input files that are needed for an experiment and information to provide them go into this section. See section 2.10, 'Defining input files for an experiment'.

### 2.3.4 Variable interpolation

The value of a .config variable may reference the value of another variable by prefixing its name with a dollar sign. This is called *interpolation* of variables. E.g.

```
# joe1234.config
WORK_ROOT = /scratch/joe
WORK_DIR = $WORK_ROOT/experiments/$EXP_ID
```

will set 'WORK\_DIR' to '/scratch/joe/experiments/joe1234'.

Interpolation only works for variables of the current section or its ancestor sections.

```
[ensembles]
  size = 42
[jobs]
  ensemble_size = $size
```

will fail with

```
Oops: missing option "size" in interpolation while reading key
'ensemble_size'
```

because 'size' is not defined in 'jobs', nor on the global level.

As in shell scripts, the variable name must be enclosed in braces if the interpolation continues with a word character (alphanumeric or underscore), or if the variable name contains a space (which is perfectly legal):

```
WORK_DIR = /tmp/$EXP_ID_test      # ERROR: missing option "EXP_ID_test"
WORK_DIR = /tmp/${EXP_ID}_test    # OK

SPACY VAR = Whew!
MESSAGE = He said: $SPACY VAR     # ERROR: missing option "SPACY"
MESSAGE = He said: ${SPACY VAR}  # OK
```

The user's *environment variables* may be referenced as global variables in a .config file. Thus a user may write something like

```
SCRIPT_ROOT = $HOME/experiments/$EXP_ID
```

setting 'SCRIPT\_ROOT' to a subdirectory of the user's home directory.

### 2.3.5 Evaluation of interpolation

While interpolation looks a lot like in shell scripts, there is a major difference: interpolation is – as in Makefiles – only evaluated when the final value is written or passed on. This has the advantage that the model setup may define settings based on variables that are only defined later in the user setup.

```
# model setup
MODEL_DIR = $HOME/$MODEL_SUBDIR
```

```
# user setup
MODEL_SUBDIR = echam
```

Here, as the model setup is read before the user setup, 'MODEL\_SUBDIR' is not set when 'MODEL\_ROOT' is defined. This works, because interpolation of 'MODEL\_ROOT's value is postponed until all levels of setup have been read.

The disadvantage is that there may be no incremental adding of values to a given variable because this would cause circular dependencies. Imagine

```
SUBMODELS = $SUBMODELS jsbach
```

When *mkexp* tries to evaluate 'SUBMODELS', it sees that it needs to do an interpolation; but to do this interpolation, 'SUBMODELS' would need to have been evaluated already! So this results in

```
Oops: interpolation loop detected in value "SUBMODELS" while reading key
'SUBMODELS'
```

### 2.3.6 Special expressions

For some applications, simply including some other variable is not enough. You might want to compute a time limit from a given constant divided by the number of computing nodes, or convert a time stamp to a list of values. For these purposes, *mkexp* includes some special expressions, that are evaluated when interpolation occurs.

```
variable = eval(expression)
variable = evals(expression)
```

Interpret *expression* as a valid Python expression and assign the result to *variable* as a string. The modules 'os', 're' and 'time' may be used in *expression*.

When the result is a list, *eval* will return a list of strings, while *evals* will return a single string, where elements are joined by a comma and a space.

Note that interpolation does not work for list values; if you need this, consider to set the original variable to a string containing a Python list expression, and then

use *eval* around the interpolation expression:

```
DATE_STRING = '[2010, 10, 20]' # need quotes here!  
DATE_LIST = eval($DATE_STRING) # becomes a 3 element list
```

```
variable = read(file_name)
```

Read the contents of the file *file\_name* and assign its contents to *variable* as a string.

```
variable = split_date(timestamp)
```

Take *timestamp* and split it into a list of numerical date/time elements. *timestamp* must have an ISO-like format (date elements separated by '-'; 'T' or space as date/time separator; time elements separated by ':'; trailing time elements and their separators are optional; time zone indicator is not supported). Unlike ISO, *split\_date* also allows the date to be in the form YYYYMMDD. Unset fields default to zero.

```
variable = sec2time(second_of_day)
```

Take integer *second\_of\_day* (from 0 to 86399) and return the corresponding time stamp as string of the form HH:MM:SS.

```
variable = 'add_years(datestamp, offset)'
```

Take integer *offset* (may be negative), add it to the year portion of *datestamp* and return the resulting date string.

```
variable = 'add_days(datestamp, offset)'
```

Take integer *offset* (may be negative), add it to the day portion of *datestamp* and return the resulting date string. Year and month portions will be set as appropriate, assuming a Proleptic Gregorian calendar with year 0.

## 2.4 Format of .tmpl files

The .tmpl files are also text files mostly written in the syntax of the job description that *mkexp* is meant to create. Currently this is usually the ksh or bash shell script syntax, but may also be any other interpreted language, like Perl or Python, or even a configuration or namelist file. The main difference are placeholders and structured comments that are embedded in the program text. These are evaluated or expanded using the information that comes with the .config files, to create the final text files, defining the jobs to be run on the target system.

The expansion of .tmpl files into the job description uses the *Jinja* Python library. It provides a default set of facilities that can be used to expand any textual template. The proposed default syntax was slightly customized to fit the needs of *mkexp*.

### 2.4.1 Placeholders

The simplest interaction in a template is replacing a template's placeholder by a value from a .config file. Any name enclosed by '%{' and '}' is taken to be a configuration variable, like in the snippets below:

```
#!/bin/ksh
# This script was created by %{NAME}
```

Here, the placeholder requests the 'NAME' variable which was set to 'Joe User' in the example .config file of section 2.3.1. This value is now looked up in the configuration and used to textually replace the placeholder expression, yielding the final text:

```
#!/bin/ksh
# This script was created by Joe User
```

## 2.4.2 Variables in sections

To request a variable within a section, simply prepend the section name to the variable name, using '.' as separator, as in

```
# %{section1.description}
```

This is also used for nested sections:

```
# %{section1.subsection1a.description}
```

For section names that contain a '.' or spaces (like 'namelist.echam' in the introductory example), instead of the '.' separator, the section name is given as a quoted string in brackets (similar to Python's dictionary syntax):

```
IS_AMIP_RUN=%{namelists['namelist.echam'].runtctl.lamip}
```

## 2.4.3 Expressions

The placeholders may also contain more complex expressions, using a limited set of operations that is defined in the *Jinja* documentation. Among these are

```
LITERAL_STRING=%{'hello'}
LITERAL_INTEGER=%{42}
LITERAL_FLOAT=%{21.5}
LITERAL_LIST=%{['hello', 42, 21.5]}
LITERAL_BOOLEANS=%{false} # Always lower-case!
ARITHMETIC=%{2 + 2 * 2 - 2 / 2} # is 5
STRING_TOGETHER=%{NAME ~ ', employee number ' ~ 42} # Converts 42 to string
LIST_ELEMENT=%{PATH[0]} # indices start with 0
LIST_SUBLIST=%{PATH[1:3]}
FILTERED_STRING=%{NAME | lower()} # is 'joe user'
```

The last example allows for a number of predefined filters instead of 'lower'. These are described in the *Jinja* documentation (List of Builtin Filters). Besides, *mkexp* defines a number of additional filters:

```
split(s, m=-1)
```

cuts the input string at all occurrences of *s*, returning a list of substrings. If *m* is positive or zero, it cuts only at the first *m* occurrences; the last element contains the remaining substring:

```
%{ 'A B C' | split(' ') }      → ['A', 'B', 'C']
%{ 'A B C' | split(' ', 1) }   → ['A', 'B C']
```

### **filter()**

removes empty elements from the input list:

```
%{ ['A', '', 'C'] | filter() }   → ['A', 'C']
```

### **match(regex, default='')**

returns the input string, if *regex* matches somewhere in it. If *regex* contains matching groups (parentheses), the substring matching the first group is returned. If no match is found, the *default* string is returned:

```
%{ 'Douglas Adams' | match('Adam') }      → 'Douglas Adams'
%{ 'Douglas Adams' | match('Eve') }        → ''
%{ 'Douglas Adams' | match('Abel', 'Cain') } → 'Cain'
%{ 'Douglas Adams' | match('l(.*)m') }     → 'as Ada'
```

### **wordwrap**(width=79, break\_long\_words=true, break\_on\_hyphens=true, wrapstring=none)

replaces the builtin *Jinja* filter 'wordwrap' by a private version that allows to suppress breaks on hyphens:

```
%{'long-hyphenated-text'|wordwrap(15, false)}
→ 'long-
   hyphenated-text'

%{'long-hyphenated-text'|wordwrap(15, false, false)}
→ 'long-hyphenated-text'
```

## 2.4.4 Loops

Text in a template may be used repeatedly, like a classical 'for' loop. Loops are defined by structured comments i.e. lines beginning with '%%'. They start with '%% for ... in ...:' and end in '%% endfor':

```
%% for countdown in [3, 2, 1, 'liftoff']:
echo {%countdown}
%% endfor
```

will be expanded by *mkexp* to yield

```
echo 3
echo 2
echo 1
echo liftoff
```

You may of course use expressions in the loop definition. For classical, index based loops, there is a 'range' function as in Python, and the size of a list is queried with the 'length' filter.



```
## for index in range(1, PATH|length()) # PATH was defined in 2.3.1
echo %{index}: %{PATH[index]}
## endfor
```

Note that indexing of lists starts at 0, i.e. the first element of the PATH list is skipped. Besides, the stop index is not included, i.e. as PATH has a length of 3, the last looping has index 2:

```
echo 1: /usr/bin
echo 2: /usr/local/bin
```

## 2.4.5 Conditions

A template may contain alternative parts that are selected depending on the .config data, similar to an 'if' statement. This is useful for e.g. skipping certain parts of the script template that are not applicable to runs of a given resolution but required for others. They are also implemented as structured comments, starting with '## if ...' and ending in '## endif', with optional '## elif ...:' and '## else:' parts.

For testing, you may use expressions with comparisons (==, !=, >, >=, <, <=), querying a certain list element (... in ...), and logical operators (and, or, not). Sub-expressions may be parenthesized to change the order of evaluation. Besides, *Jinja* provides a number of named tests that use the '... is ...' Syntax.

```
## if PATH|length() is divisibleby 3:
diff3 %{PATH[:3]|join(' ')}
## elif PATH|length() is even:
diff %{PATH[:2]|join(' ')}
## else:
echo cannot handle PATH
## endif
```

If PATH is defined as in the examples above, this will result in

```
diff3 /bin /usr/bin /usr/local/bin
```

For a list of available tests, see the *Jinja* documentation.

## 2.4.6 Comments

*Jinja* also allows template comments that are removed when the template is expanded. This is implemented as another kind of structured comment starting with '##%#':

```
# This comment will make it to the expanded script
##%# This one will not make it and is for template documentation only
```

## 2.4.7 Block statements and block comments

For templates that contain more *Jinja* code than actual output lines, a variant of the

standard *Jinja* block syntax is available for both statements and comments<sup>1</sup>. Block statements begin with '{%\_\_mkexp\_\_' and end with '%}', block comments begin with '{#\_\_mkexp\_\_' and end with '#}'.

## 2.5 Standard experiments

When generating an experiment setup, *mkexp* expects the .config and .tmpl files to reside in a subdirectory of the current working directory, called 'standard\_experiments'.

The definition of a standard experiment type *typename* may consist of a configuration in *typename.config* and a number of *typename.jobname.tmpl* files, one for each subsection *jobname* of the jobs section. Before reading *typename.config*, the special DEFAULT.config is loaded, containing the model default settings. Both .config and .tmpl files may be missing; the default is to read only DEFAULT.config or the corresponding DEFAULT.jobname.tmpl file instead.

The name of an experiment type may be of the form *experimentkind-experimentquality*, as in amip-LR above. In this case, the experiment type is supposed to be of a certain *quality*, like a given model resolution (LR), but to share the overall experiment structure with all types of the same *kind* (amip). Therefore, the .config files take the full name, *experimentkind-experimentquality.config*, whereas the templates are defined as *experimentkind.jobname.tmpl*, independent of the requested quality.

An experiment configuration must contain the special variable EXP\_TYPE, set to the name of experiment type to use.

## 2.6 Standard options

Besides the standard experiment types, *mkexp* also supports option sets that are independent of the experiment type chosen.

Usually these option sets contain a number of settings needed for a certain technical aspect, e.g. for changing the output interval or aggregation method for output data, or providing resolution dependent model settings. They reside in a subdirectory 'standard\_options' of the current working directory, each in their respective *optionname.config* file.

Within the experiment's .config file, options are selected by setting the variable EXP\_OPTIONS to the list of required option names. These settings are loaded after the experiment type configuration but before the user defined experiment configuration.

---

1 The standard *Jinja* comment syntax '{#' collides with the Bourne shell idiom for variable size, '\${#var}'. Standard block statement syntax '{%' gives problems when mkexp template variables '%{var}' are used in shell variable expansions, like '\${%{var}}:-default}'

## 2.6.1 Options set due to model configuration

Some options may need to be set for all experiments that use a given model configuration. If e.g. a part of the model is disabled at build time, the corresponding option set should also be disabled for all experiments.

For this, the build process may write an optional file 'SETUP.config' that is read before any type or user configuration. If this file contains the variable `SETUP_OPTIONS`, the options listed there will be loaded before loading the `EXP_OPTIONS` list. Do not override `SETUP_OPTIONS` in the user configuration unless you know what you are doing!

## 2.7 Generating jobs

When running *mkexp*, the special configuration section `[jobs]` is read and evaluated. Each of its subsections, e.g. `[[run]]`, defines a job definition file or job script to be created.

```
# model setup: experiment type 'control'
[jobs]
  [[pre]]
  [[run]]
  [[post]]
```

For each of the jobs defined in the model setup above, there must be a template file in the model setup, e.g. for `[[run]]` either as 'control.run.tmpl' or 'DEFAULT.run.tmpl'. The corresponding file is expanded to its final form using the full experiment configuration, as described before. Besides, the job specific variables are set and passed according to their respective template.

The resulting job scripts are written to the directory defined by `SCRIPT_DIR`, e.g. as 'joe1234.run', and marked as being executable. Besides, as mentioned before, the contents of the special variable `EXP_DESCRIPTION` is written to a `README` file in that same directory. Also, an update script is created that allows to re-generate all output files with identical environment and command line settings by simply running './update' from the script directory.

### 2.7.1 Changing the model job list

Usually, the job list is defined in the model setup. The user may chose to add jobs and delete jobs from this list as appropriate. While adding a job is straightforward, removing a job uses a special section variable '.remove' (note the leading period). It is defined in the `[jobs]` section and contains a list of the jobs to be suppressed.

```
# joe1234.config
EXP_TYPE = control
[jobs]
.remove = post, pre
[[my_pre]]
[[my_post]]
```

This way, the 'pre' and 'post' jobs will not be created in favor of two new jobs, 'my\_pre' and 'my\_post'. In this case, the user setup must provide two templates 'joe1234.my\_pre.tmpl' and 'joe1234.my\_post.tmpl', together with the .config file.

Alternatively, you may want to introduce a new 'my\_post' job, that is basically the same as the old 'post' job but uses a slightly different configuration. This may be done using the special section variable '.extends'.

```
[jobs]
[[my_post]]
.extend = post
command = $HOME/bin/my_special_command
```

With this configuration, an additional 'joe1234.my\_post' is created based on the existing 'post' template. A dedicated 'joe1234.my\_post.tmpl' file is not needed here. The 'command' setting is made available to the template via the 'JOB' dictionary.

## 2.7.2 Pre-defined job variables

While *mkexp* in general does not impose any naming convention on the variable names used in the job sections and leaves the details to the respective model setup, there are a few exceptions.

### tasks

The total number of parallel (MPI) tasks that will be started when running the model. If a job section does not define 'tasks', its value defaults to 'nodes' times 'tasks\_per\_node'. Some models require that 'tasks' may explicitly be set to some artificial value to trigger the testing mode.

### nodes

Number of computing nodes required on the computing system. Needed if 'tasks' is not set.

### tasks\_per\_node

Number of parallel (MPI) tasks on a single node. Needed if 'tasks' is not set.

## 2.7.3 Overriding namelist settings in derived jobs

There is special provision to change namelist files settings for a specific job. Consider this setting from the introductory example.

```
[namelists]
  [[namelist.jsbach]]
    [[[jsbach_ctl]]]
      use_dynveg = false
```

If – for some reason – your experiments needs 'use\_dynveg' set to 'true' for the first year only, you may create an additional 'run\_first' job, with a job specific namelists subsection that – apart from the additional brackets – has the same structure as the global namelists section.

```
[jobs]
  [[run_first]]
    .extends = run
    [[[namelists]]]
      [[[[namelist.jsbach]]]]
        [[[[[jsbach_ctl]]]]]
          use_dynveg = true
```

This will result in a 'joe1234.run\_first' file that is identical to 'joe1234.run' except for the 'use\_dynveg' setting.

## 2.7.4 Native script variables

While the definition of .config variables may use variable references like \$NAME or \${NAME} to include the verbatim value of other .config variables, this may not always be what you want. If you want to create a job script that is supposed to be 'user-serviceable' for certain applications, the users will not appreciate having to change the same value several times in the same script. Instead they will want to have a single, native script variable that is used throughout the job script, and that may be re-defined on a single line.

To allow this, *mkexp* locates all expressions like \${NAME} in the configuration values, and re-formats them to the syntax of the current job script:

```
# joe1234.config
NAME = Joe User
MESSAGE = This experiment was generated by ${NAME}

### joe1234.job.tpl
#!/bin/sh
NAME='${NAME}'
echo ${MESSAGE}
```

By default, native variables are formatted as shell script, namely \${NAME}:

```
#!/bin/sh
NAME='Joe User'
echo This experiment was generated by ${NAME}
```

To support variable references for other script languages, a job specific variable '.var\_format' may be defined. It defines an output format string where any occurrence of '%s' will be replaced by the respective variable name. For a Python based script, this

may look like:

```
# joe1234.config
NAME = Joe User
MESSAGE = This experiment was generated by ${NAME}
[jobs]
  [[job]]
    .var_format = '' + str(%s) + ''

### joe1234.job.tmpl
#! /usr/bin/env python
NAME = '${NAME}'
print('${MESSAGE}')
```

This setup will expand to

```
#! /usr/bin/env python
NAME = 'Joe User'
print('This experiment was generated by ' + str(NAME) + '')
```

## 2.7.5 Initializing native script variables

In the previous section, the native variables were initialized by an additional script line. While this is sufficient for a small number of variables, it may be difficult to maintain these initialization lines for a more complex setup with changing requirements.

To allow a self-maintaining variable list based on the current configuration, *mkexp* maintains the special variable `VARIABLES_`. When generating output for

```
NAME = Joe User
EMAIL = joe@domain.tld
MESSAGE = This experiment was generated by ${NAME} <${EMAIL}>
```

*mkexp* will parse all values, recognize 'NAME' and 'EMAIL' as native variables, and will put their names in the `VARIABLES_` list. Now we may use the 'for' template directive to generate an additional line for each member of `VARIABLES_`. To query the value for a given variable name, *mkexp* provides the 'context' function, such that the template

```
#! /bin/sh
### for variable in VARIABLES_:
  %{variable}='${context(variable)}'
### endfor
echo "%{MESSAGE}"
```

eventually yields

```
#!/bin/sh
EMAIL='joe@domain.tld'
NAME='Joe User'
echo "This experiment was generated by ${NAME} <${EMAIL}>"
```

Note that the order of variable names is not necessarily the order in which they were defined in the `.config` file.

## 2.7.6 Re-generation of scripts and backup

Even the simplest user setup may contain an error. In this case, it is considered good practice to fix this error in the user setup, and to rerun *mkexp*. If you want to be really good, you might even start a new experiment from the previous one's restart data.

As a convenient short-cut, *mkexp* provides the script 'update' in the script directory. It may be called instead of going back to the 'run' and re-running *mkexp* directly. The update script records all command line settings and environment settings that were used for running *mkexp* so that './update' regenerates the scripts exactly as they were created, without having to re-construct the exact settings. Like *mkexp*, update allows to set or change variables on the command line, e.g.

```
./update FINAL_DATE=2015-12-31
```

may be used to regenerate scripts with a new final date. Note that these settings are also recorded, i.e. running just './update' the next time will again set `FINAL_DATE`.

Of course, there is also the possibility to change the *generated* job scripts directly, and then go on. This is fine as long as the required setting is using a native variable or is otherwise easily editable. On the other hand, facing the next change, this might not be the case, so eventually you may need to re-generate the whole thing. What now about those manual changes to the job scripts?

Whenever *mkexp* sees existing job scripts while trying to generate the new description, it will automatically create backup files. They are placed in a subdirectory of `SCRIPT_DIR`, named 'backup'. Also, the scripts for each regeneration are bundled in their own subdirectory, named after the current date-time stamp. This way you may easily compare old and new scripts to evaluate and possibly transfer any manual changes after a necessary regeneration.

## 2.8 Standard environments

The above examples neglect an important feature of job descriptions, namely the system or machine dependent set up. Different computing centers use different job control software and naming conventions, might provide different versions of the same software at differing locations. This needs to be handled in a way that is independent of the other configuration as far as possible.

For this, a standard environment, like a standard experiment, consists of a `.config` file

and a corresponding template. The *environmentname.config* file contains settings like directory paths or a description of machine capacities for job control

The actual job control headers needed to run a certain job step are saved as *environmentname.tmpl*. This template will be filled using configuration information from both experiment, options, and environment. The resulting job header is usually included at the beginning of the experiment's job script templates.

An experiment configuration may set the special variable `ENVIRONMENT` to the name of the host environment to use. If it is not set, or empty, the 'DEFAULT' environment settings will be used.

## 2.9 Defining namelists and other configuration files

Most models need at least one Fortran namelist file or another form of configuration file to run. The special section `[namelists]` is designed to contain all information that goes into these files. Each immediate subsection defines settings for a single file that will by default be formatted as a Fortran namelist. For example,

```
[namelists]

  [[namelist.echam]]
    [[runctl]]
      lamip = true
      nproma = 48
      earth_angular_velocity = 7.3e-5
      out_expname = joe1234
      dt_stop = 2009, 1, 1, 0, 0, 0
```

defines a single namelist file, 'namelist.echam', containing a single namelist group with four variables of different types. Note that you do not need to use quotes for strings nor periods for logical values.

### 2.9.1 Formatting the namelist information

The names of the subsections of each namelist file entry, i.e. the second level subsections under the immediate subsections of `[namelists]`, are taken as namelist group names, and their variables are formatted as fields of this namelist group. In the example above, there is a single namelist group 'runctl', which will be converted to the Fortran namelist convention. The values of the group's fields are checked to determine whether they are numerical, logical, or string values. Logical and string values will then be formatted using periods or quotes, respectively.

In the example above, the first four fields are single values of logical, integer, floating point, and string type. The fifth is a list of integer values, that will be passed as such to the namelist file:



```

&runcntl
  lamip = .true.
  nproma = 48
  earth_angular_velocity = 7.3e-5
  out_expname = 'joe1234'
  dt_stop = 2009, 1, 1, 0, 0, 0
/

```

Please note that both group and field names are converted to lower case! Two fields named 'key' and 'Key' will result in two 'key = ...' lines, and will cause trouble. We recommend using lower case for all group and field names.

## 2.9.2 Suppressing namelist groups or variables

If you want to disable a namelist group defined on a higher setup level, you may set the special group variable '.hide' to 'true'. As you may suspect, setting '.hide' to 'false' for namelist groups that were hidden on a higher setup level will re-enable them.

To disable a single namelist variable, simply omit the value after the equals sign. This is taken to be an empty string, causing the variable to not be written to the namelist file, so the default value is used instead. Setting the variable to a non-empty value in a higher level .config file will re-enable it.

If the empty string is a valid value in your namelist, you may set the special variable '.default' to an alternative string to be used as default marker instead of the empty string. This can be done both on namelist and group level.

As a legacy, the namelist group section also honors the special variable '.remove' which may be set to a list of names. Any variables listed in the '.remove' variable will be deleted from the namelist group definition. Note that, while '.remove' is useful to suppress default settings that must not be present for the experiment setup, currently there is no way to resurrect a removed setting in a later setup level. Therefore, its use in model setups is strongly discouraged.

## 2.9.3 Comments in namelists

In general, '#' comments in namelist or group sections will be converted to Fortran 90 '!' comments. In-line comments for individual settings are also recognized.

```

# Run time settings
[[[runcntl]]]
  lamip = true # Use AMIP conventions
  # Block size for optimization
  nproma = 48

```

```

! Run time settings
&runctl
    lamip = .true. ! Use AMIP conventions
    ! Block size for optimization
    nproma = 48

```

In addition, *mkexp* recognises namelist settings that were commented out. For these, the formatting of values as described before is also carried out within the comment.

```

# lamip = true # Use AMIP conventions

```

```

! lamip = .true. ! Use AMIP conventions

```

There is one hitch: if the last setting in a group is commented, the *configobj* library will take this line to be a group comment for the following group. To work around this, *mkexp* recognises a special namelist variable '.end' to keep the commented setting with the first group.

```

[[[parctl]]]
    # nprocb = 48 # Ends up with 'runctl' instead
[[[runctl]]]
    # lamip = true # Stays with 'runctl'
    .end =
[[[dynctl]]]

```

```

&parctl
/
! nprocb = 48 ! Ends up with 'runctl' instead
&runctl
    ! lamip = .true. ! Stays with 'runctl'
/
&dynctl

```

## 2.9.4 Using the namelist text

The formatted namelist text is stored in a global variable that may be used by a template placeholder. This way, all job information available is written to a single script or description file, and native script variables may be used in the namelist definitions.

The name of this variable is generated from its respective file name, like 'namelist.echam', by converting all letters to upper case (namelist.echam → NAMELIST.ECHAM) and replacing non-word characters by an underscore (NAMELIST.ECHAM → NAMELIST\_ECHAM).

```

# joe1234.config
[namelists]
    [[namelist.echam]]
        [[runctl]]
            out_expname = ${EXP_ID}

```

```

#!/bin/sh
EXP_ID=%{EXP_ID}
cat > namelist.echam << EOF
%{NAMELIST_ECHAM}
EOF

```

which expand to

```

#!/bin/sh
EXP_ID=joe1234
cat > namelist.echam << EOF
&runctl
    out_expname = '${EXP_ID}'
/
EOF

```

Note how this setup uses the native script variable 'EXP\_ID' to set the namelist contents.

As an option, the namelist text may be formatted using the global function 'format\_namelist', taking the namelist section as argument.

```

#!/bin/sh
EXP_ID=%{EXP_ID}
cat > namelist.echam << EOF
%{format_namelist(namelists['namelist.echam'])}
EOF

```

The result will be the same as in the original example.

Additionally, 'format\_namelist' can take the name of a group within the namelist as a second argument, allowing to format groups individually.

```

#!/bin/sh
EXP_ID=%{EXP_ID}
cat > namelist.echam << EOF
%{format_namelist(namelists['namelist.echam'], 'runctl')}
EOF

```

In this special case the result will still be the same, as 'runctl' is the only group within 'namelist.echam'.

As shown in the examples above, the script template is responsible for writing the namelist text to an actual file. While in general the namelist file takes the same name as the .config subsection defining it, *mkexp* will not enforce this. The template needs to be set up accordingly.

## 2.9.5 Using native script variables in namelists

When using native script variables in a namelist, it may be necessary to suppress the conversion of values to namelist conventions. Consider

```
[namelists]
  [[namelist.echam]]
    [[[runctl]]]
      dt_stop = $$final_date

final_date='2015, 12, 31, 23, 52, 30'
cat > namelist.echam << EOF
%{NAMELIST_ECHAM}
EOF
```

When the text for NAMELIST\_ECHAM is generated, *mkexp* converts the value of 'dt\_stop' to a string surrounded by single quotes (see 'EXP\_ID' in the previous examples), as '\$\$final\_date' cannot be recognized as a numeric or logical value. In the namelist context however it is needed as an unquoted list of integers. To fix this, simply use the special syntax 'raw(...)' around the value:

```
dt_stop = raw($$final_date)
```

which disables the conversion to a valid namelist value, leaving the correct formatting of the native variable to the script.

## 2.9.6 Non-namelist configuration files

A [namelists] subsection may not only define a Fortran namelist file but also a custom format configuration file. For any of these files, the special section variable '.use\_template' may be set to 'true', if the model setup provides a template '*subsection.tmpl*', or to an arbitrary template name, replacing *subsection*, if a template is applicable to more than one section. This template is expanded using the subsection's variables to create a suitably formatted text. As for genuine namelists, the result is stored in a global variable.

## 2.10 Defining input files for an experiment

While the namelist files determine the model properties, the model state also depends on input files for initial and boundary conditions or assimilation data. These files are defined in the [files] special section.

Each subsection may define or override one of two special variables:

.base\_dir

file names are taken to be relative to this directory

.sub\_dir

file names are taken to be relative to this subdirectory of '.base\_dir'

For expanding the resulting file path, *mkexp* provides two global functions:

```
get_file(section, name)
```

returns the full path for file *name* as given in the *section* object. Note that *section* is given as object reference, but *name* is given as a string. If the value of *name* is an absolute file name, it will override .base\_dir and .sub\_dir. If it contains native

script variables, they are expanded to their top-level configuration values; if none is found, the native script variables are formatted as described before.

```
get_dir(section)
```

returns the directory for the given *section* object

Other than that, there are no restrictions on the content structure of this section, and the job script templates are responsible for converting this content into a suitable script text.

### 2.10.1 Overriding input files for certain jobs

As for namelists, there is special provision to change the [files] settings for a specific job. Let us assume that a model run continues a previous run and reads that run's state from a restart file that may be defined like this:

```
[files]
  [[echam]]
    [[[restart]]]
      restart_myexp_echam.nc = restart_myexp_echam_18491231.nc
```

The first run will instead pick up its state from another experiment. This is done with a job specific 'files' subsection that – apart from the additional brackets – has the same structure as the global 'files' section.

```
[jobs]
  [[run_first]]
    [[[files]]]
      [[[[echam]]]]
        [[[[[restart]]]]]
          restart_myexp_echam.nc = restart_anotherexp_echam_18491231.nc
```

This will result in a '.run\_first' script that gets the model state from 'anotherexp', while the '.run' script uses 'myexp'.