Joint UK Land Environment Simulator (JULES)

Version 2.0

User Manual

Last revised: DBC 03 Sept 2007.

1. Introduction and what's new

The Joint UK Land Environment Simulator (JULES) is a computer model that simulates many soil and vegetation processes. It is based on the Met Office Surface Exchange System (MOSES), the land surface model used in the Unified Model of the UK Met Office.

This document describes how to run version 2.0 of JULES. It primarily describes the format of the input and output files, and does not include detailed descriptions of the science and representation of the processes in the model.

Further information can be found on the JULES website http://www.jchmr.org/jules.

What's new in this version

The physical processes and their representation in version 2.0 have not changed from version 1. However, version 2.0 is much more flexible in terms of input and output, and allows JULES to be run on a grid of points. New features include:

- Ability to run on a grid.
- Choice of ASCII or binary formats for input and output files (also limited support of netCDF input).
- More flexible surface types – number and types can vary.
- Optional time-varying, prescribed vegetation properties.
- More choice of meteorological input variables.
- Optional automatic spin up.
- Enhanced diagnostics – large choice of variables, frequency of output, sampling frequency, etc.

2. Overview of JULES

This section provides a brief overview of JULES, largely so as to provide background information and introduce terms used in the rest of the manual. Further details on the science and process descriptions contained in JULES can be found at the JULES website http://www.jchmr.org/jules.

JULES views each gridbox as consisting of a number of surface types. The fractional area of each surface type is either prescribed by the user or modelled by the TRIFFID sub-model. Each surface type is represented by a tile, and a separate energy balance is calculated for each tile. The gridbox average energy balance is found by weighting the values from each tile. In its standard form, JULES recognises nine surface types: broadleaf trees, needleleaf trees, C3 (temperate) grass, C4 (tropical) grass, shrubs, urban, inland water, bare soil and ice. These 9 types are modelled as 9 tiles. A land gridbox is either any mixture of the first 8 surface types, or is land ice. Note that, with version 2.0, one is not limited to these 9 standard surface types.

Soil processes are modelled in several layers, but all tiles lie over and interact with the same soil column. Each gridbox requires meteorological driving variables (such as air temperature) and variables that describe the soil properties at that location. It is also possible to prescribe certain characteristics of the vegetation, such as Leaf Area Index, to vary between gridboxes.

JULES can be run for any number of gridboxes from one upwards. The number of gridboxes is limited by the availability of computing power and suitable input data. When run on a grid, JULES models the average state of the land surface within the area of the gridbox and most quantities are taken to be homogeneous within the gridbox (with options to include subgrid-scale variability of a few, such as rainfall). In that case, the input data are also area averages. JULES can also be run "at a point", with inputs that are taken to represent conditions at that point – this configuration might be used when field measurements of meteorological conditions are available.

3. Building and running JULES

Building a JULES executable requires two pieces of software:

- a Fortran 90 compiler
- a version of the 'make' utility

It may also be desirable, but not essential, to have available the following software:

- the Fortran 90 netCDF[1] interface library

3.1. The make utility

The Makefile supplied with the JULES code should be compliant with most versions of make, but is only guaranteed to work with GNU Make[2] (also known as gmake), which is available on most Linux and UNIX systems and also for Windows. Once the Makefile is set up for the user's system, JULES is built simply by entering 'make' at the command prompt while in the directory containing the Makefile. This will compile all of the JULES source code, make a library libjules.a, and finally link the compiled source to create and executable file with a default name of jules.exe. To remove all the files created during the build process enter 'make clean' at the command prompt.

The make utility uses architecture- and compiler-specific variables that must be set by the user to the appropriate values for their system. These values may be set in the files Makefile.common.mk and Makefile.comp.*. (The user should not have to edit the file named Makefile.) There are two convenience options, COMPILER and BUILD, which should be passed to make from the command line to tell that program where the appropriate values should be taken from. The COMPILER option allows the user to define a list of compiler-specific variables (including compiler flags) without having to edit the Makefile. The BUILD option allows the user to build with sets of predefined flags for different situations, e.g. debugging. The Type and permitted values for each of these options are described in Table 1, and additional information is given in the comments at the head of Makefile.

The compiler-specific variables are specified in individual files named Makefile.comp.* for a handful of common compilers, e.g. Makefile.comp.sun. The list of tested compilers includes two (Intel and G95) that can be downloaded for no cost via the URLs in Table 1. To use a compiler that is not listed, the user should replace the '@@' strings in the blank compiler file Makefile.comp.misc with values appropriate to their compiler and invoke make with the option COMPILER=misc.

[1] The netCDF interface library can be downloaded for no cost from http://www.unidata.ucar.edu/software/netcdf/
[2] The GNU Make utility can be downloaded for no cost from http://www.gnu.org/software/make/

Table 1. Options that can be passed to `make` when building JULES.

Variable	Type and permitted values	Notes
`COMPILER`	`sun`	Default option; use options for Sun Studio compiler series (previously known as Workshop and Forte).
	`intel`	Use options for Intel Fortran compiler for Linux, Windows and MacOS (http://www3.intel.com/cd/software/products/asmo-na/eng/compilers/284132.htm). Version 9.0 was used for testing and it was found that two lines in the source code had to be changed – find these and the suggested replacements by searching the code for "Intel".
	`g95`	Use options for G95 compiler, (http://www.g95.org/).
	`nag`	Use options for NAGWare compiler.
	`pgf`	Use options for Portland Group compiler.
	`misc`	Use options for an unlisted compiler.
`BUILD`	`run`	Default option; for normal compilation of JULES.
	`debug`	Switch on compiler debug flags.
	`fast`	Switch on compiler optimisation flags for faster execution.
`CDFDUMMY`	`false`	Use a precompiled netCDF library.
	`true`	Use the dummy netCDF library provided with JULES.

3.2. The netCDF interface library

To build JULES, the user must also pass `make` some information about the netCDF interface library. If the user has access to a pre-compiled netCDF interface library, then they should pass `make` the options `CDF_LIB_PATH` and `CDF_MOD_PATH`. The values for these options are the directories in which the pre-compiled netCDF library (`libnc.a`) and Fortran 90 module files (those with `.mod` extension) are located respectively. This can be done also by editing the `Makefile` itself, but the recommended method is by specifying the variables as options when `make` is invoked, e.g., '`make CDF_LIB_PATH=$HOME/mynetcdf/lib CDF_MOD_PATH=$HOME/mynetcdf/mod`'.

If the user does not have access to a pre-compiled netCDF library, then JULES may be compiled by specifying '`CDFDUMMY=true`' when `make` is invoked rather than setting the `CDF_LIB_PATH` and `CDF_MOD_PATH` variables. This option compiles a set of dummy netCDF interface functions, which merely allows the rest of the JULES code to compile correctly and provides no functionality. **When this option is used JULES will neither read from nor write to netCDF format files**. The user must ensure that netCDF input/output options are not selected at any point in any JULES control file (described in Section 6) used with an executable produced using this option.

3.3. Example build lines

To build JULES using the normal Sun compiler options and link with a netCDF library:

```
make COMPILER=sun BUILD=run CDF_LIB_PATH=$HOME/mynetcdf/lib \
CDF_MOD_PATH=$HOME/mynetcdf/mod
```

To build JULES using the fast Intel compiler options and do not link with a netCDF library:

```
make COMPILER=intel BUILD=fast CDFDUMMY=true
```

These command lines can become quite long and tedious to keep typing, so it's a good idea to set the list of frequently used ones as environment variables:

```
export JULESBUILD=”COMPILER=sun BUILD=run \
CDF_LIB_PATH=$HOME/mynetcdf/lib \
CDF_MOD_PATH=$HOME/mynetcdf/mod”

make $JULESBUILD
```

It is then possible to override options specified in that variable by adding revised ones at the end:

```
make $JULESBUILD BUILD=debug
```

3.4. Running JULES

A JULES executable is run by redirecting standard input to a file that contains all the information needed to describe a run, e.g.,

```
jules.exe < run1.jin
```

The format of this input file is described in Section 6, with some example runs described in Section 6.20.

The file extension ".jin" is meant to suggest "**J**ULES **I**nput **F**ile", but there is no need to use this or any other extension.

4. Overview of the JULES code

The general structure of the JULES source code, including the order in which routines are called, is illustrated below. For the sake of clarity, the full details are not shown here. In particular, the initialisation and output steps (subroutines `init` and `output`) can call several routines.

```
jules--|
       |--init--|
       |        |--init calls various initialisation routines
       |
 (top of timestep loop)
       |
       |--drive_update
       |
       |--veg_update
       |
       |--control---|
       |            |--tile_albedo--|
       |            |               |--albpft
       |            |               |--albsnow
       |            |--sf_expl--|
       |            |           |--tilepts
       |            |           |--physiol--|
       |            |           |           |--root_frac
       |            |           |           |--smc_ext
       |            |           |           |--raero
       |            |           |           |--sf_stom--|
       |            |           |           |           |--qsat
       |            |           |           |           |--canopy--|
       |            |           |           |           |          |--leaf_c3
       |            |           |           |           |          |--leaf_c4
       |            |           |           |--soil_evap
       |            |           |           |--leaf_lit
       |            |           |           |--cancap
       |            |           |           |--microbe
       |            |           |--heat_con
       |            |           |--sf_exch--|
       |            |                       |--qsat
       |            |                       |--sf_orog
       |            |                       |--sf_resist
       |            |                       |--sf_rib_sea
       |            |                       |--sf_rib_land
       |            |                       |--sf_orog
       |            |                       |--fcdch_sea
       |            |                       |--  phi_m_h_sea
       |            |                       |--fcdch_land
       |            |                       |--  phi_m_h_land
       |            |                       |--sf_resist
       |            |                       |--dustresb
       |            |                       |--  vgrav
       |            |                       |--sf_flux_sea
       |            |                       |--sf_flux_land
       |            |                       |--stdev1_sea
       |            |                       |--stdev1_land
       |            |                       |--sf_orog_gb
       |            |                       |--sfl_int_sea
       |            |                       |--   phi_m_h_sea
       |            |                       |--sfl_int_land
```

```
      |            |                       |--   phi_m_h_land
      |            |--sf_impl--|
      |            |           |--im_sf_pt
      |            |           |--sf_evap
      |            |           |--sf_melt
      |            |           |--screen_tq--|
      |            |           |             |--qsat
      |            |           |--sice_htf
      |            |
      |            |--hydrol---|
      |            |           |--sfsnow
      |            |           |--surf_hyd--|
      |            |           |            |--frunoff
      |            |           |            |--sieve
      |            |           |            |--pdm
      |            |           |--calc_baseflow
      |            |           |--soil_hyd--|
      |            |           |            |--hyd_con(_vg)
      |            |           |            |--darcy(_vg_--|
      |            |           |            |              |-- hyd_con(_vg)
      |            |           |            |--gauss
      |            |           |--calc_fsat
      |            |           |--soil_htc--|
      |            |           |            |--heat_con
      |            |           |            |--gauss
      |            |           |--ice_htc
      |            |           |--soilmc
      |            |           |--soilt
      |            |           |--ch4_wetl
      |            |--veg2--|
      |            |        |--tilepts
      |            |        |--phenol
      |            |        |--triffid--|
      |            |        |           |--vegcarb--|
      |            |        |           |           |--growth
      |            |        |           |--lotka--|
      |            |        |           |         |--  compete
      |            |        |           |
      |            |        |           |--soilcarb--|
      |            |        |           |            |--decay
      |            |        |--tilepts
      |            |        |--sparm--|
      |            |                  |--pft_sparm
      |            |--veg1--|
      |                     |--tilepts
      |                     |--phenol
      |                     |--sparm--|
      |                               |--pft_sparm
      |--output
      |
(bottom of timestep loop)
      |
      |--jules_final
```

5. File formats and the JULES grid

5.1. Overview of file formats

JULES aims to support input and output in three formats: ASCII, netCDF and a generic binary format (simply called 'binary' below). At present the ASCII and binary options are implemented, but only a very limited implementation of input via netCDF exists[3], and there is no netCDF output. The netCDF options will be improved for future releases. Input can also be read from many PP files (a format used by the UK MetOffice). The binary format is compatible with the GrADS[4] package, amongst others. A run control file might indicate that data are to be read from several files, using one or more of these file formats. For example, soil data might be in an ASCII file, while meteorological driving data are in netCDF files.

A "self-describing file" (SDF) is one in a format that contains metadata describing the contents of the file. For JULES, only a netCDF file is presently considered to be a SDF. Minimal use is made of any metadata contained within a file, including SDFs and PP files. For example, a SDF might contain data that describes the grid or the times of data, but these are not used by JULES. Instead, this information is provided via the run control file and all input data must be provided on the same grid.

For all non-SDF files, the data model is based on that used by GrADS. Each variable is viewed as being 4-dimensional in (x, y, z, t) on a regular grid. Although we will talk of x and y in terms of West-East and South-North compass directions, in general the grid can be any rectilinear grid, with West-East being replaced by "left to right". X varies in the direction from West to East, y varies from South to North (this default can be changed), and z varies from bottom to top. All variables in any one file must have the same grid size in x and y (i.e. all variables are on a grid of `nx*ny` points), and have a value at all times (although that value could indicate a missing datum). The data are stored as a series of xy slices, with x varying fastest, then y, then z, and t varying slowest. For example, say we have a file with two variables (A and B) on a grid with `nx=2`, `ny=2`. A has `nz=1`, and B has `nz=2`. In the JULES/GrADS model, the data must be stored in the input file in the order,

```
A(x=1,y=1,z=1,t=1)  # 1st xy plane of A at t=1
A(x=2,y=1,z=1,t=1)
A(x=1,y=2,z=1,t=1)
A(x=2,y=2,z=1,t=1)
B(x=1,y=1,z=1,t=1)  # 1st xy plane of B at t=1
B(x=2,y=1,z=1,t=1)
B(x=1,y=2,z=1,t=1)
B(x=2,y=2,z=1,t=1)
B(x=1,y=1,z=2,t=1)  # 2nd xy plane of B at t=1
B(x=2,y=1,z=2,t=1)
B(x=1,y=2,z=2,t=1)
B(x=2,y=2,z=2,t=1)
A(x=1,y=1,z=1,t=2)  # 1st xy plane of A at t=2
```

[3] In brief, input via netCDF has been enabled for vector input from files with certain netCDF dimensions. This allows GSWP2 data held at CEH to be used. A user who is familiar with the netCDF library will be able to modify the existing code to read other netCDF files.

[4] The GrADS software can be downloaded for no cost from http://grads.iges.org/grads/gadoc/index.html

```
A(x=2,y=1,z=1,t=2)
… etc …
```

For clarity, this example has shown each datum on a separate line, but in fact any number of data within a single field (see below) can be on the same line.

A data "field" is considered to be a single x-y plane of data (i.e., `nx*ny` values). Header records can be present at the start of a file, at the start of each time within the file, and at the start of each field.

Note that this means that JULES reads and writes data in terms of 'maps' (all values of one field, then all values of another field), rather than using a 'point-by-point' data model (all fields for one point, then all fields for another point).

A related concept used in JULES, is that of the point number in input or output files. This is used to select individual points from a larger grid. The point number runs from 1 at the gridpoint in the SW corner of the grid, across rows (so the bottommost row contains gridpoints 1 to `nx`), and then from South to North up the grid. Examples and further discussion of JULES grids can be found in Section 6.4.

5.2. Describing the format of a file

Variables that describe how data are arranged in files are used in several sections later in this document. These variables are summarised in Table 2. Often the information that JULES will read and use from the control file depends on the file format of any one data file. The information required for ASCII, binary or PP files is generally fairly similar, while netCDF files are rather different.

Table 2. Format of frequently used control file options.

Variable name	Type	Notes
`readFile`	logical	Switch that indicates source of data. TRUE: data are read from a named, external file FALSE: data are read from the run control file
`fileFormat`	character	Flag indicating the file format. Case sensitive. Only used if `readFile=.TRUE.` `'asc'`: ASCII `'bin'`: generic binary (including GrADS) `'nc'`: netCDF `'pp'`: PP format

5.2.1. ASCII or binary files

If `fileFormat='asc', 'bin' or 'pp'` or `'pp'` some or all of the following information is read from a section that starts with the tag '`>ASCBIN`'. Exactly what information is needed varies between cases (for example, it is assumed that there is a single time "level" in a file of soil properties, so `nheaderTime` is not needed).

Table 3. Format of options used to specify the reading of ASCII, binary and PP format files.

Variable name	Type	Notes
`nheaderFile`	integer	The number of header records at the top of a file. For an ASCII file, a header record is a line in the file. For a binary file, a header record is an individual word or record (e.g. a single 'real' value). Not used for a PP file.
`nheaderTime`	integer	The number of header records that precedes the data for each time level within a file. Not used for a PP file.
`nheaderField`	integer	The number of header records that precedes each field (x-y plane) of data. Not used for a PP file.
`fieldNumber`	integer	This is used to locate a given field (xy plane) within all the fields available at each time level. If there are `nFieldFile` fields of data at each time level, and `fieldNumber=2` for a particular variable, the second field of data is used for this variable.

Blank lines between fields in an ASCII input file can cause the code to read the wrong data, and should be avoided. If blank lines are present between fields, they should be interpreted as header lines.

There are restrictions on what PP files JULES can read. Each field must have no trailing "extra data" (i.e. `header(20)` must be zero). It is also assumed that the data are ordered as in the JULES/GrADS model outlined above (so, for example, we do NOT have all times of field 1, then all times of field 2), so that the required data can be found without using the information contained in the field headers. The headers are used to check that the size of the field and the STASH code are as expected. The STASH code for each variable is currently hardwired in the code. At the time of writing the PP-reading code has no known bugs, but it has been used much less than other options, so any more obscure bugs might not have been triggered.

5.2.1.1. An example ASCII input file

Table 4 shows part of an example ASCII file that could be read by JULES, with `nheaderFile=2`, `nheaderTime=1`, `nheaderField=1`. The size of the input grid is assumed to be `nxIn=3`, `nyIn=2`. There are 2 variables, A which has a single level, and B which has 2 levels, giving a total of 3 fields per time. Annotation after any "!" (and shown in *italics*) would NOT be present in the actual file. The data are shown on 2 lines per field, but this is not important – `nx*ny` values will be read however they are presented.

Table 4. Part of an example ASCII file that could be read by JULES.

```
This file contains example data.                      !  1st file header
There are 2 variables, the 2nd with 2 levels.         !  2nd file header
Time level 1.                                         !  header for time=1
Variable A                                            !  header for 1st field
12.0   15.6    17.1                                   !  data for A at t=1
-1.0   23.9    53.2
Variable B, level 1                                   !  header for 2nd field
22.0   25.6    12.1                                   !  data for B at t=1, 1st level
-1.0   22.9     23.2
Variable B, level 2                                   !  header for 3rd field
32.0   11.6    12.1                                   !  data for B at t=1, 2nd level
-9.1   72.9     43.7
Time level 2.                                         !   header for time=2
Variable A                                            !   header for 1st field
9.2     67.3    -7.6                                  !   data for A at t=2
11.5   23.9    -8.3
Variable B, level 1                                   !   header for 2nd field
---- rest of file not shown ---
```

5.2.2. netCDF files

If `fileFormat='nc'`, the following information is read from a section that starts with the tag '`>NC`'. As noted earlier, support for netCDF input is currently rather limited in JULES.

Table 5. Format of options used to specify the reading of netCDF format files.

`Option name`	Type	Notes
`nDim`	integer	The number of dimensions used for a variable in the netCDF file. This was set at the time the netCDF file was created.
`dimName(1:nDim)`	character	The names of the netCDF dimensions for a variable.
`SDFname`	character	The name used in the netCDF file for the chosen variable.

6. The JULES control file

6.1. Introduction

Each run of the JULES code is controlled by a text file that is called the "run control file". Broadly speaking, the run control file holds three types of information:

- the general details of the run, such as start and end dates
- the values for parameters of the model, such as albedo
- the specification of the required output

The JULES code is designed to be moderately flexible, in that there are switches that allow the user to select between different configurations, and it can accommodate input data in several different file formats. This flexibility means that the run control file may be relatively long and the user has to check that all values are set correctly. The documentation below aims to help the user in this task. Example input files can be found as described in Section 6.20.

The run control file has a particular format, in that the lines must be in a particular order and must contain various headers. The file is read by various routines arranged under the subroutine INIT, using FORTRAN list-directed input [i.e. the format is given as "*" in a `READ` statement of the form `READ(unit,*)`]. The JULES executable is run with standard input redirected to this control file, e.g. `jules.exe < control_file.jin`. The use of list-directed input means that there may be more than one arrangement of input values that can be read by the code – for example a single line with 10 values or 2 lines with 5 values each. Repeated numerical values can often be specified using the "*" notation (e.g. 100 values of 1.0 can be entered as `100*1.0`), which can sometimes be useful in specifying a large but constant field.

"Tags" are used to indicate the start of each section, and allow the code to skip directly to this point ignoring any intermediate lines. Each tag is of the form,
`>SECTION_NAME`
and must be included exactly as in the example run control files, using capital letters and with no space before or after the initial >.
These section tags are listed in Table 6.

Table 6. Sections in a JULES control file.

Section name	Description	Described in manual section
`INIT_OPTS`	General model options.	6.2
`INIT_TIME`	Start and end times for simulation, timestep lengths, spin up.	6.3

`INIT_GRID`
`INIT_LAND`
`INIT_LATLON`
	Set up the model grid.	6.4

`INIT_FRAC`	Set gridbox tile fractional coverage options.	6.5
`INIT_DZSOIL`	Set soil vertical level options.	6.6
`INIT_SOIL`	Set model soil parameters.	6.7
`INIT_VEG_PFT`	Set uniform parameters for vegetation tiles.	6.8
`INIT_VEG_VARY`	Set parameters for vegetation tiles that vary in	6.9

`INIT_VEG_VARY2`	either space or time.	
`INIT_NONVEG`	Set parameters for non-vegetation tiles.	6.10
`INIT_SNOW`	Set snow related parameters.	6.11
`INIT_TRIF`	Set parameters for TRIFFID dynamic vegetation model.	6.12
`INIT_AGRIC`	Set fraction of each gridbox that is agriculture for use with TRIFFID.	6.13
`INIT_MISC`	Set miscellaneous carbon-cycle parameters.	6.14
`INIT_DRIVE`	Set input driving data options.	6.15
`INIT_IC`	Set initial conditions of all prognostic variables.	6.16
`INIT_OUT`	Set options for model output.	6.17
`NEWPROF`	Set up an output profile.	6.17

The user can annotate the run control file, for example to add comments, but these must not interfere with the reading of the rest of the file. Depending upon the details of the run, there are various locations in which it is "safe" to include annotation, but the only really safe location is on the lines immediately preceding a "tag" (described above). Annotation can also often be placed on the same line as the data at the end of any data field (i.e. so that the code reads the values required and will not see the annotation).

Values of character variables, such as file names, should be enclosed within quotation marks (either single ' ' or double " "). Character variables have a maximum length specified in the code, which are sometimes given in this documentation, e.g. character*8 indicates a variable of length 8. Logical values can be entered in any of the formats understood by FORTRAN, e.g. T, true or `.TRUE.` may all be used to represent true. In the sections beow, the sizes of certain arrays are indicated using brackets: e.g. `myArray(1:20)` requires values for the 20 elements numbered 1 to 20.

Although a spatial field can be read from the run control file, in practice this becomes unwieldy for large grids, and most spatial fields are likely to be stored in separate files, the names of which can be listed in the run control file.

In the following sections, the first column lists the variables that are to be read from a line, and subsequent columns give the type and a brief description of each variable. The variable names given are generally those used to declare the corresponding FORTRAN variables (except where the code uses temporary workspace and a more meaningful variable name is given in this documentation).

6.2. `INIT_OPTS`: General model options

```
>INIT_OPTS

npft, nnvg
ntiles
pftName(1:npft)
nvgname(1:nnvg)

nxIn, nyIn
sm_levels
can_model
can_rad_mod, ilayers
l_cosz, l_spec_albedo
l_phenol, l_triffid, l_tif_eq

yrevIn
echo
print_step
```

Table 7. Description of options in `INIT_OPTS` section.

Variable name	Type and permitted values	Notes
`npft`	integer >=1	The number of plant functional types to be modelled.
`nnvg`	integer >=1	The number of non-plant surface types to be modelled. The total number of surface types to be modelled is called `ntype`, and is given by `ntype=npft+nnvg`. In the standard setup, JULES models 5 vegetation types and 4 non-vegetation types (`npft`=5, `nnvg`=4). However, the model domain need not contain all 9 types – e.g. the domain could consist of a single point with 100% grass. The amount of each type in the domain is set in the section `INIT_FRAC` (Section 6.5).

`ntiles`	integer `npft+nnvg` or `1`	The number of tiles for each gridbox. This is the number of surface energy balances that are solved for each gridbox. `ntiles=ntype`: a separate energy balance is calculated for each surface type. `ntiles=1`: aggregate parameter values are used to solve a single energy balance per gridbox. Generally `ntiles=ntype` is preferred, but `ntiles=1` is used by the Met Office forecast model where computational cost must be kept to a minimum. When using TRIFFID, `ntiles` must equal `ntype`=9.
`pftName(1:npft)`	character array	Names of PFTs. When JULES looks for parameter values for the PFTs, it looks for these names.
`nvgName(1:nnvg)`	character array Must include `'soil'`.	Names of non-vegetation surface types. When JULES looks for parameter values for the surface types, it looks for these names.
`nxIn`	integer >=1	The number of columns of data in the input grid (see further discussion of the grid in Section 6.4).
`nyIn`	integer >=1	The number of rows of data in the input grid.
		The total number of points in the input grid is thus `nxIn*nyIn`. If the input data consists of a single point, `nxIn=nyIn=1`. A vector of points is specified by setting `nyIn=1`. Although the notation may suggest a regular, rectangular grid, the model can be run at any number of arbitrary locations, the most likely way of doing so being to set `nyIn=1`, `nxIn`=number of points.
`sm_levels`	integer >=1	Number of soil layers. A value of 4 is often used.
`can_model`	integer 1, 2, 3 or 4	Choice of canopy model for vegetation: 1: No canopy. 2: Radiative canopy with no heat capacity. 3: Radiative canopy with heat capacity. 4: As 3 but with a representation of snow beneath the canopy. • NB `can_model=1` does *not* mean that there is no vegetation canopy. It means that the surface is represented as a single entity, rather than having distinct surface and canopy levels for the purposes of radiative processes.

`can_rad_mod`	integer 1, 2 or 3	Switch for treatment of canopy radiation. 1: Beer's law for light penetration and "big leaf" for scaling leaf to canopy scale photosynthesis 2: Multi-layer approach for both radiation interception (2-stream approach of Sellers et al., 1992) and canopy photosynthesis. 3: As 2, but photosynthesis calculated separately for sunlit and shaded leaves. References: Sellers, P. et al., 1992, Remote Sens. Environ., 42: 187-216. Jogireddy, V.R. et al., 2006, Hadley Centre technical note 63. Available from http://www.metoffice.gov.uk/research/hadleycentre/pubs/HCTN/index.html Mercado, L.M., et al., 2007, Tellus, 59B 553-565
`ilayers`	integer ≥1	Number of layers for canopy radiation model. Only used if can_rad_mod=2 or 3. These layers are used for the calculations of radiation interception and photosynthesis.
`l_cosz`	logical	Switch for calculation of solar zenith angle. For land points, this switch is only relevant if `l_spec_albedo=TRUE` (otherwise it is better set to FALSE to prevent unnecessary calculations). TRUE: calculate zenith angle. FALSE: assume constant zenith angle of zero, meaning sun is directly overhead.
`l_spec_albedo`	logical	Switch for albedo model. TRUE: use spectral albedo. This includes a prognostic snow albedo. FALSE: use a single (averaged) waveband albedo.
`l_phenol`	logical	Switch for vegetation phenology model. TRUE: use phenology model. FALSE: do not use phenology model.
`l_triffid`	logical	Switch for dynamic vegetation model (TRIFFID). TRUE: use TRIFFID. FALSE: do not use TRIFFID.
`l_trif_eq`	logical	Switch for equilibrium vegetation model (i.e., an equilibrium solution of TRIFFID). This is only used if `l_triffid=TRUE`. TRUE: use equilibrium TRIFFID. FALSE: do not use equilibrium TRIFFID.

`yrevIn`	logical	Switch indicating if the order of the rows in the input data is not the JULES standard. TRUE: Input data are arranged in North to South order (i.e. first data are from northernmost row). FALSE: Input data are arranged in South to North order (the JULES standard). Note that this does not affect how JULES numbers points on its internal grids – within JULES the numbering always runs from South to North. This switch applies to all input files.
`echo`	logical	Switch controlling output of messages to standard output (e.g. screen). TRUE: print messages to screen. This will print the values of parameters, and also print messages when files are opened or closed. This is useful while checking that a run is correctly set up, but can result in a large volume of data if the model grid is large. FALSE: suppress printing of most messages to screen
`print_step`	integer >=1	The number of timesteps in between the printing of timestep information. Every `print_step` timesteps, the model prints the current timestep number and date to standard output. While this can be a useful way to follow the progress of a model integration, frequent messages can generate a large amount of unnecessary output during long integrations.

6.3. INIT_TIME: Date and time information

```
>INIT_TIME

timestep
dateMainRun(1), timeRun(1), dateMainRun(2), timeRun(2)

dateSpin(1:2), nspin
spinFail
spinFlag(1), spinTolPercent(1), spinTol(1)
spinFlag(2), spinTolPercent(2), spinTol(2)

phenol_period, triffid_period
l_360
```

Table 8. Description of the `INIT_TIME` section

Variable name	Type and permitted values	Notes
`timestep`	integer >=1	Timestep length (seconds). A typical timestep is 30 to 60 minutes. If the timestep is too long, the model becomes numerically unstable.

`dateMainRun(1:2)`
`timeRun(1:2)`
	integer array, character*8 array	These specify the start and end times for the integration. Each run of JULES consists of an optional spin-up period and the "main run" that follows the spin up. See below for more about the specification of the spin up. For simplicity, the same times of day are used for both the spin-up and main periods.
		The main run starts at `timeRun(1)` on `dateMainRun(1)` and ends at `timeRun(2)` on `dateMainRun(2)`.
		Dates should be given in format yyyymmdd. All dates must be >0. Times should be given in format hh:mm:ss. All times are in Greenwich Mean Time (UTC) – but see 6.3.1

`dateSpin(1:2)`	integer array	The dates for the spin-up period, in the format yyyymmdd. Elements 1 and 2 are the start and end dates respectively. The spin-up phase of the integration must be over times that either, immediately precede the main run. In this case the spin-up phase is from `timeRun(1)` on `dateSpin(1)` to `timeRun(1)` on dateSpin(2) [where dateSpin(2) equals dateMainRun(1)]

		OR are the same as those for the main run. In this case the spin-up phase is from timeRun(1) on dateMainRun(1) to timeRun(2) on dateMainRun(2). Examples are given below.
`nspin`	integer >=0	The maximum number of times the spin-up period is to be repeated: 0: no spin up >0: at least 1 and at most `nspin` repetitions of spin up are used. After each repetition, the model tests whether the selected variables have changed by more than a specified amount over the last repetition (see below). If the change is less than this amount, the model is considered to have spun up, and the model moves on to the main run.
`spinFail`	logical	Switch controlling behaviour at the end of spin up period, if the model has not passed the spin-up test. Only used if `nspin`>0. TRUE: End the run if model has not spun up. FALSE: Continue the run.
`spinFlag(1:2)`	logical array	Switch indicating whether the variable is included in the decision as to when spin up is achieved. TRUE: include this variable in analysis of spin up FALSE: exclude this variable The 2 elements refer to: 1: moisture content of each soil layer (kg m^{-2}) 2: temperature of each soil layer (K)
`spinTolPercent(1:2)`	logical array	Switch indicating whether the tolerance for this variable is expressed as a percentage. TRUE: tolerance is a percentage FALSE: tolerance is an absolute value The elements refer to different variables – see `spinFlag`.
`spinTol(1:2)`	real array	Tolerance for spin up. For each spin-up variable, this is the maximum change over a repetition of spin up that is allowed if the model is to be considered as spun-up. If the absolute value of the change (or the magnitude of the percentage change if `spinTolPercent = TRUE`) is less than or equal to `spinTol`, the variable is considered to have spun up. For example, `spinTol=0.1` means that the variable in question must change by less than 0.1 over a cycle of spin up if it is to be considered spun up. Spin up is assessed using the difference between instantaneous values at the end of consecutive cycles of spin up. For example, if the spin up period is from

		15 Jan 2005 to 15 Jan 2006, every time the model gets to 15 Jan 2006 the spin-up variables are compared with their value at the end of the previous cycle.
`phenol_period`	integer >=1	Period for calls to phenology model (days). Only relevant if `l_phenol`=TRUE.
`triffid_period`	integer >=1	Period for calls to TRIFFID model (days). Only relevant if one of `L_TRIFFID` is TRUE.
`l_360`	logical	Switch indicating use of 360 day years. TRUE: each year consists of 360 days. This is sometimes used for idealised experiments. FALSE: each year consists of 365 or 366 days.

6.3.1. Note on time convention

If a run requires that the solar zenith angle be calculated (`l_cosz`=TRUE), then times must be in Greenwich Mean Time (UTC), so that the code can calculate the zenith angle at each location and time. However, if `l_cosz`=FALSE, the user might prefer to use Local Time, particularly if this is used for input or validation data, as the timestamp on model output will then match that on the other data.

6.3.2. Examples of dates and times

1. *A run without spin up*

```
19970101,'00:00:00', 19990101,'01:00:00'  !  dateMainRun, timeRun
19970101,19970102,0                       !  dateSpin, nspin
```

This specifies a run from midnight on 1st January 1997 until 01:00 GMT on 1st January 1999. `nspin=0` means there is no spin up.

2. *A run with spin up over a period that immediately precedes the main run*

```
19970101,'00:00:00', 19990101,'01:00:00'  !  dateMainRun, timeRun
19960101,19970101,5                       !  dateSpin, nspin
```

This specifies a spin-up period from midnight on 1st January 1996 to midnight on 1st January 1997 (the time of day is taken from the first line). This spin-up will be repeated up to 5 times, before the main run from midnight on 1st January 1997 until 01:00 GMT on 1st January 1999.

3. *A run with spin up over a period that overlaps the main run*

```
19970101,'00:00:00', 19990101,'01:00:00'  !  dateMainRun, timeRun
19970101,19980101,5                       !  dateSpin, nspin
```

This specifies a spin-up period from midnight on 1st January 1997 to midnight on 1st January 1998 (the time of day is taken from the first line). This spin-up will be repeated up to 5 times, before the main run from midnight on 1st January 1997 until 01:00 GMT on 1st January 1999.

4. Example of specifying requirements for spin up

```
T               !  terminate run if spin-up fails (T,F)
T, F, 1.0       !  soil moisture: spinVar,spinTolPercent, spinTol
T, T, 0.1       !  Tsoil
```

The first value, `spinFail`=TRUE, means that if the spin-up has not "converged" after `nspin` cycles, the run will end. Convergence is measured using moisture content and temperature of each soil layer. At every point and in every layer, soil moisture must change by less than 1 kg m-2 (or mm of water), while soil temperature must change by less than 0.1%.

6.3.3. Notes on spin up

Note that at present the analysis of whether the model has spun up or not is limited to aspects of the "physical" state of the system, and does not explicitly consider carbon stores, making it less useful for runs with interactive vegetation (TRIFFID).

During the spin-up phase of a run, the JULES code provides the correct driving data (for example, meteorological data) as the model time "cycles" round over the spin up period. Consider the case of a spin up over 1 Jan 2005 to 31 Dec 2005. At or near the end of 31 Dec 2005 during the spin up, the driving data will start to adjust to the values for 1 Jan 2005. The calculated driving data may vary slightly between the start or end of the first cycle and similar times in later cycles, because of the need to match the data at the end of each cycle to that at the start of the next cycle. Generally this does not cause a problem.

Depending upon the details of the input data and any temporal interpolation, the driving data may vary rapidly at the end of a cycle of spin up, causing an extreme response from the model. In most cases the model will adjust, possibly with large heat fluxes over a few hours, but the user should be aware that unusual behaviour near the end/start of a spin up cycle may be the result of this adjustment. Consider the case of a spin up over 1 Jan 2005 to 31 Dec 2005. At or near the end of 31 Dec 2005 during the spin up, the driving data will start to adjust to the values for 1 Jan 2005, which could be very different from conditions on 31 Dec 2005. The length of time over which the driving data adjust depends on the frequency of the data, and the choice of temporal interpolation. For example, with 3-hourly data that is interpolated onto a one hour timestep, the adjustment will take place over 3 hours. However, hourly data and an hourly timestep will force an instantaneous adjustment at the start of 1 Jan 2005.

Although `nspin` specifies the *maximum* number of spin up cycles, some of which might not be used if the model is considered to have spun up earlier, it is possible to specify the exact number of cycles that will be performed. This can be done by demanding an impossible level of convergence by setting `spinTol`<0 (remember that `spinTol` is compared with the *absolute* change over a cycle) and setting `spinFail=FALSE` so that the integration continues when spin up is judged to have failed after `nspin` cycles.

Although it is expected that a spin up phase will be followed by the main run in the same integration, it is possible to do the spin up and main run in separate integrations. This can be done by demanding an impossible level of convergence by setting `spinTol`<0, setting `spinFail=TRUE` so that the integration stops when spin up is judged to have failed, and setting `dumpFreq` (see Section 6.17.1) to any value that writes a final dump. The final state of the model, after `nspin` cycles of spin up, will be written to the final dump, and a subsequent simulation started from this dump.

A limitation of the current code is that it cannot cope with a spin up cycle that is short in comparison to the period of any input data. For example, a spin up cycle of 1 day cannot use 10-day vegetation data. The code will likely run but the evolution of the vegetation data will probably not be what the user intended! However, it is unlikely that a user would want to try such a run.

Occasionally, the model fails to diagnose a spun up state when in fact the integration has reached a quasi-steady state that is not detected by the procedure of assessing spin up through comparison of instantaneous values at the end of consecutive cycles of spin up. An example of this is "period-2" behaviour, where the model state repeats itself over a period of 2 cycles. Such behaviour should be apparent in the model output during spin up, and the user can opt to repeat the integration over a given number of spin up cycles, and not to wait for a spun-up state to be diagnosed.

6.4. Grid description

The process of setting up the model grid involves three parts of the run control file: `INIT_GRID`, `INIT_LAND` and `INIT_LATLON`.

`INIT_GRID` is used to select how the model grid will be specified, e.g. all points within a given range of latitude and longitude.
`INIT_LAND` is used to set a land/sea mask.
`INIT_LATLON` specifies the latitude and longitude of each point.
These three sections are then followed by the `DATA_POINTS`, `DATA_LAND` and `DATA_LATLON` sections which provide input data (if that is to come from the run control file).

Each run of JULES involves two grids: the input grid, and the model grid. The input grid is the grid on which all input data are held. The model grid is the set of points on which the model is run. The model grid is a subset of the input grid.

The JULES grid is a rectangle of size `nx*ny` points, including the special case of `ny`=1 when the grid is a vector of points. The points to be modelled may be selected from a larger input grid, by specifying one or more of (1) a list of point numbers (2) a range of latitude and longitude (3) that only land points are to be selected. The grid may contain both land and sea points, although sea points can be omitted. A vector of points can be used to select locations that are not adjacent in the real world - for example, one might only want to run the model at locations within a catchment for which observations are available. In this case although the model could be run on a grid that included the whole catchment, it is more efficient to run only at the selected points.

6.4.1. `INIT_GRID`: setting up the grid

```
>INIT_GRID

pointsList, landOnly
subArea, subAreaLatLon
xcoord(1:2),ycoord(1:2)
npoints
readFilePoints
fileNamePoints
```

Table 9. Description of options in the `INIT_GRID` section.

Variable name	Type and permitted values	Notes
`pointsList`	logical	Switch indicating whether the model grid is to be specified as a list of points. **TRUE: Points to be modelled are selected from the input grid via a list of points.** In this case, the points to be modelled are selected via a list of point numbers (see

		below). Land fraction is not read. **FALSE: All points in the input grid are modelled – subject to elimination by `subArea` or `landOnly` (see below).** Land fraction is read. Land points to be modelled are indicated by land fraction>0, sea points by land fraction<=0. A sub-area can be selected (see below). The value of `npoints` is set by the model, and equals the number of land points. Although this field was originally intended to be a land/sea mask (1,0) , more generally it can now be thought of as a map of points to be modelled, e.g. "land fraction" can be set to 1 at all locations within a catchment, and to zero (or less) at all other points (such as land points outside the catchment).
`landOnly`	logical	Switch indicating if only land points are to be modelled. If `pointsList=TRUE`, `landOnly` must be FALSE. **TRUE: Only land points are modelled. Sea points are excluded from the model grid.** **FALSE: All points are modelled (land and sea).**
`subArea`	logical	**TRUE: a subsection of the input grid will be used** (see `xcoord` and `ycoord` below) **FALSE: the full input grid is considered.**
`subAreaLatLon`	logical	If `subArea=TRUE`, this indicates how to interpret the coordinates `xcoord` and `ycoord`. **TRUE: co-ordinates are longitude and latitude.** **FALSE: co-ordinates are x and y indices (column and row numbers).**
`xcoord(1:2)`	real array	x-coordinates of the sub-area to be considered. Depending on `subAreaLatLon`, these are longitudes (in range -180 to 360^o) or column numbers. See notes on grid definition in Section 6.4. If values are column numbers, the code uses the nearest integer to the input value.
`ycoord(1:2)`	real array	As `xcoord`, expect in latitudinal (y) direction.
`npoints`	integer	The number of points that are to be modelled. Only used if `pointsList`=TRUE.
`readFilePoints`	logical	Switch controlling source of list of point numbers. Only used if `pointsList=TRUE`. **TRUE: read from an external ASCII file** **FALSE: read from the run control file**. Points are specified at the sub-section marked `>DATA_POINTS` (see Section 6.4.3).
`fileNamePoints`	character	Name of file containing list of points. Only used if

		pointsList=TRUE.

6.4.2. INIT_LAND: Land fraction

This section describes how the land fraction field is set. Land fraction describes the fraction of each gridbox that is land.

```
>INIT_LAND

readFileLand
fileFormatLand
fileNameLand

>ASCBIN
nheaderFileLand,nheaderFieldLand
fieldLand

>NC
nlandDim
landDim(1:nlandDim)
varNameLand
```

Table 10. Description of options in the INIT_LAND section.

Variable name	Type and permitted values	Notes
readFileLand	logical	Switch controlling source of land fraction data. Only used if pointsList=FALSE. TRUE: read from an external file FALSE: read from the run control file, at the section marked >DATA_LAND (see Section 6.4.3).
fileFormatLand	character See Section 5.2.	Format of file containing land fraction data.
fileNameLand	character	Name of file containing land fraction data.
The following are read only if readFileLand=TRUE. Only values for the appropriate file format are read.		
>ASCBIN: If fileFormatLand='asc','bin' or 'pp':		
nheaderFile	integer >=0	The number of headers at the start of the land fraction file. See Section 5.2.
nheaderField	integer >=0	The number of headers before each field in the land fraction file. See Section 5.2.
fieldLand	Integer >=1	The field number in the file that holds data for the first level of this variable. See discussion of fields in Section 5.1.
>NC: If fileFormatLand='nc':		

`nLandDim`	integer >=1	Land mask data in a SDF is held in an array of `nLandDim` dimensions.
`landDim(1:nlandDim)`	character array	Names of land mask dimension variables in SDF.
`varNameLand`	character array	The name of the variable containing the land fraction.

6.4.3. `INIT_LATLON`: Latitude and longitude

```
>INIT_LATLON

regLatLon
regLat1, regLon1
regDlat, regDlon
readFileLatLon, fileFormatLatLon
fileNameLatLon

>ASCBIN
nheaderFile, nheaderField
fieldLat, fieldLon

>NC
nLatLonDim
latLonDim
varNameLat,varNameLon

>DATA_POINTS
pointList(1:npoints)
>DATA_LAND
flandg(1:nxIn,1:nyIn)
>DATA_LATLON
latitude(1:nxIn,1:nyIn)
longitude(1:nxIn,1:nyIn)
```

Table 11. Description of options in the `INIT_LATLON` section.

Variable name	Type and permitted values	Notes
`regLatLon`	logical	Switch indicating if the input grid is ‘regular’ (and will be described by origin and increment) or if latitude and longitude fields are to be read. TRUE: the grid is ‘regular' and can be specified by its origin and gridbox size. There is then no need to read lat/lon values for each gridpoint. FALSE: read latitude and longitude values for each gridpoint.
`regLat1`	real	The latitude (decimal degrees North) of the southernmost row of gridpoints in the input grid (NOT

		necessarily the model grid). The gridpoint is in the centre of the gridbox.
`regLon1`	real -180 to 360	The longitude (decimal degrees East) of the westernmost column of gridpoints in the input grid (NOT necessarily the model grid).
`regDlat`	real >0.0	The size of a gridbox in the NS direction (decimal degrees of latitude). Note: `regLat1` and `regLon1` are only used if `regLatLon`=TRUE. `regDlat` and `regDlon` may be used even if `regLatLon`=FALSE, if there is a need to establish the area of each gridbox (which is needed by some parameterisations and to label output).
`regDlon`	real >0.0	The size of a gridbox in the EW direction (decimal degrees of longitude).
`readFileLatLon`	logical	Switch controlling source of latitude and longitude data. Only used if `pointsList`=FALSE and `regLatLon`=FALSE. TRUE: read from an external file FALSE: read from the run control file, at the section marked >DATA_LATLON.
`fileFormatLatLon`	character	Format of file containing latitude and longitude data.
`fileNameLatLon`	character	Name of file containing latitude and longitude data.
The following are read only if `readFileLatLon=TRUE`. Only values for the appropriate file format are read.		
>ASCBIN: If `fileFormatLatLon='asc', 'bin' or 'pp'`:		
`nheaderFile`	integer >=0	The number of headers at the start of the lat/lon file. See Section 5.2.
`nheaderField`	integer >=0	The number of headers before each field in the lat/lon file. See Section 5.2.
`fieldLat`	integer >=1	The field number in the file that holds latitude data. See discussion of fields in Section 5.1.
`fieldLon`	integer >=1	The field number in the file that holds longitude data.
>NC: If `fileFormatLatLon='nc'`:		
`nLatLonDim`	integer >=1	Lat/lon data in a SDF are held in arrays of `nLatLonDim` dimensions.
`latLonDim(1:nlatLonDim)`	character array	Names of dimension variables in SDF.
`varNameLat`	character	The name of the variable containing the latitude data.
`varNameLon`	character	The name of the variable containing the longitude data.
The following sections are used only if the switches above indicate that the fields are to be read from the run control file.		
`pointList(1:npoints)`	integer array >=1	A list of the points that are to be modelled. These are point numbers in the input grid. NB If the input data run from North to South (i.e. not the JULES S to N order), the point numbers should still be

		calculated following the JULES S to N convention. Thus point number 1 is in the SW corner of the grid, which will not be the first point in the input data if `yrevIn`=T (unless `nyIn`=1).
`flandg(1:nxIn, 1:nyIn)`	real array	The fraction of each gridbox that is land.
`Latitude(1:nxIn,1:nyIn)`	real array	The latitude of each gridpoint.
`Longitude(1:nxIn,1:nyIn)`	real array -180 to 360	The longitude of each gridpoint. All values should be in the range of either -180 to 180º or 0 to 360º.

The special case of an equal angle grid (all gridboxes have same extent in terms of latitude and longitude) in which the rows run WE and the columns SN (hereafter referred to as an equal angle grid), can be set up via a simple option. All other grids, including a vector of points, require the latitude and longitude of all points to be input.

If `regLatLon`=TRUE, the input data must be presented in the default JULES order (starting bottom left at (`regLat1,regLon1`) and proceeding row-wise). If `regLatLon`=FALSE, the input data need not be in order of lat/lon coordinates – each point in the grid will use the lat/lon read in for that point.

6.4.4. Examples of grid description

The latitude and longitude of the grid must be specified for all runs. For many model runs, the location of the grid is important, since it controls important factors such as the angle of the sun. Other, more idealised, runs might not need this information, but in this case the location may still be required so that the model output can be correctly mapped. If the location is not needed for either purpose, the user should enter an arbitrary location (e.g., 0°N, 0°E).

Grid example 1: A single point run.

This covers the simplest case: the input contains a single point. We assume that `nxIn=1` and `nyIn` =1 (see Section 6.2). All values are obtained from the run control file – no other file is involved. Only the lines in **bold** are relevant, and irrelevant sections have been omitted.

```
>INIT_GRID

T,F                !  pointsList, landOnly
F,F                !  subArea, subAreaLatLon
1,2,3,4            !  xcoord(1:2),ycoord(1:2)
1                  !  npoints
F                  !  readFilePoints
'points.txt'       !  fileNamePoints

>INIT_LAND
F                  !  readFileLand
'bin'              !  fileFormatLand
'grid.gra'         !  fileNameLand
```

```
>INIT_LATLON
T                !  regLatLon
40.0, 50.0       !  regLat1, regLon1
1.0,1.0          !  regDlat, regDlon
F                !  readFileLatLon
'bin'            !  fileFormatLatLon
'latlon.gra'      !  fileNameLatLon

>DATA_POINTS
1                 !  pointList

>DATA_LAND
1.0               !  flandg

>DATA_LATLON
0.0               !  latitude
5.0               !  longitude
```

`pointsList=T` indicates that the grid will be described by a list of points.
`npoints=1` shows that this run is for a single point.
`readFilePoints=F` indicates that the point numbers are read from the `>DATA_POINTS` section, where point number 1 is indicated (the only possibility for an input grid of one point).
`readFileLand=F` indicates that the land fraction field is read from the `>DATA_LAND` section, where the value 1.0 shows that the single gridbox is 100% land.
`regLatLon=T` indicates that the grid is 'regular' and will be described by its origin (`regLat1`, `regLon1`) and gridbox size (`regDlat`, `regDlon`). There is then no need for any further information about coordinates – in particular the data at `>DATA_LATLON` are not read.

Grid example 2: Selecting points in a given range of latitude and longitude.

The grids used in this example are shown in Figure 1. The input grid has `nxIn=5`, `nyIn=4`, and we wish to model the area 55-57ºN 355-357ºE (5ºW-3ºW). To do this, we use the following entries in the run control file. Only the lines in **bold** are relevant, and irrelevant sections have been omitted.

```
>INIT_GRID

F,F                     !  pointsList, landOnly
T,T                     !  subArea, subAreaLatLon
355.0,357.0,55.0,57.0   !  xcoord(1:2),ycoord(1:2)
1                       !  npoints
F                       !  readFilePoints
'points.dat'            !  fileNamePoints

>INIT_LAND
T                 !  readFileLand
'bin'             !  fileFormatLand
'grid.gra'        !  fileNameLand
```

```
>ASCBIN
0,0              !  nheaderFileLand,nheaderFieldLand
1                !  fieldLand

>INIT_LATLON
T                !  regLatLon
55.5, 353.5      !  regLat1, regLon1
1.0, 1.0         !  regDlat, regDlon
F                !  readFileLatLon
'bin'            !  fileFormatLatLon
'grid.gra'       !  fileNameLatLon
```

`pointsList`=F indicates that the model grid will be determined by the land fraction field (and also latitude and longitude in this case).
`landOnly`=F indicates that both sea and land points will be selected.
`subArea`=T indicates a sub-section of the input grid is requested. `subAreaLatLon`=T indicates that the sub-section will be specified by a range of latitude and longitude, shown by `xcoord` and `ycoord` to be 355ºE to 357ºE,55·0ºN to 57·0ºN (note we could enter the longitude range as -5 to -3).
`npoints` is irrelevant because the number of points will be determined as the number of points the model finds within the given lat/lon range.
`readFileLand`=T indicates that the land fraction field is read from the binary file called 'grid.gra', which has no headers and contains land fraction as the first field.
`regLatLon`=T indicates that the input grid is a 'regular' grid, with origin (the gridpoint in the southwest corner) shown by regLat1, regLon1 to be 55.5ºN 355.5ºE, and gridbox size 1º×1º.

With this information, JULES determines that there are 4 gridpoints within the given lat/lon range, and that the model grid will be a square of side 2 gridboxes. The land fraction field shows that these are all land points, meaning that the land vector also has 4 points. Note that these points could also have been selected by providing a list of the point numbers, indicated by `pointsList=TRUE`, `npoints=4`, and then entering the point numbers (3, 4, 8, 9) after `>DATA_POINTS`.

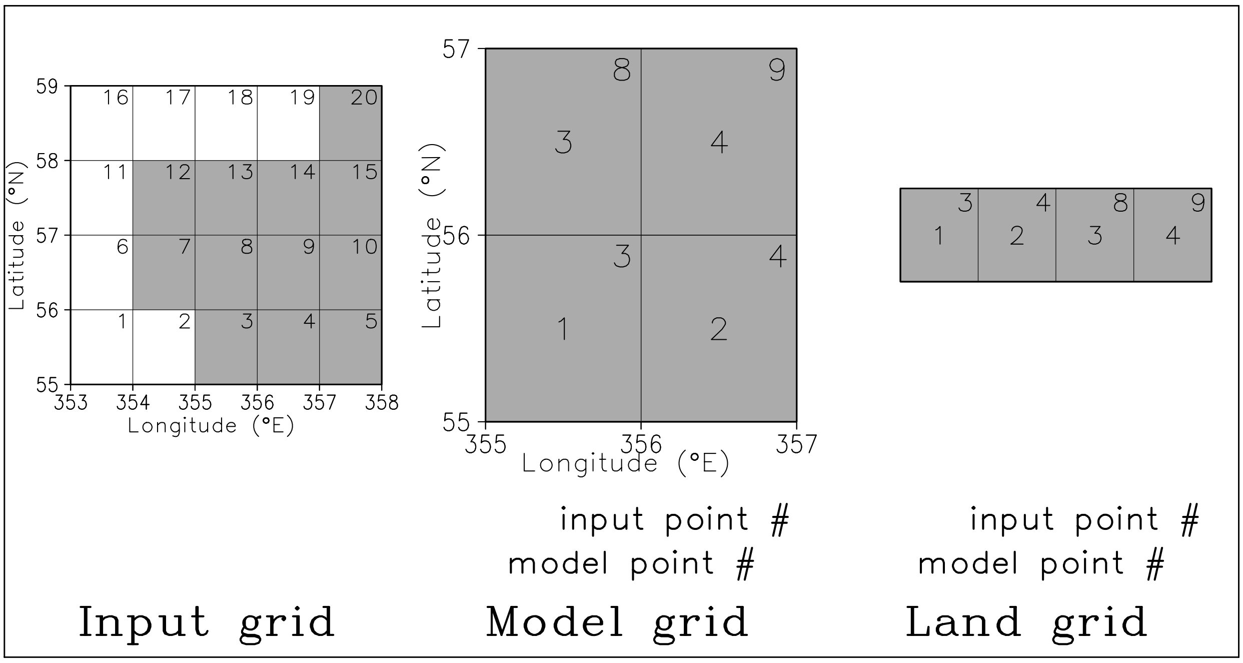

Figure 1. Example of grid selection based on longitude and latitude.

Grid Example 3: Selecting only land points in a given range of latitude and longitude.

This example is similar to Example 2, but this time we only wish to model land points within a given area. The grids used in this example are shown in Figure 2 and we wish to model land points in 55-57ºN 354-356ºE (6ºW-4ºW).

To do this, we use the following entries in the run control file. Only the lines in **bold** are relevant, and irrelevant sections have been omitted.

```
>INIT_GRID

F,T                      !  pointsList, landOnly
T,T                      !  subArea, subAreaLatLon
-6.0,-4.0,55.0,57.0      !  xcoord(1:2),ycoord(1:2)
1                        !  npoints
F                        !  readFilePoints
'points.dat'             !  fileNamePoints

>INIT_LAND
T                   !  readFileLand
'bin'               !  fileFormatLand
'grid.gra'          !  fileNameLand
```

`pointsList=F` indicates that the model grid will be determined by the land fraction field (and also latitude and longitude in this case).

`landOnly`=T indicates that only land points will be selected.
`subArea`=T indicates a sub-section of the input grid is requested. `subAreaLatLon`=T indicates that the sub-section will be specified by a range of latitude and longitude, shown by `xcoord` and `ycoord` to be 6ºW to 4ºE, 55ºN to 57ºN.
`npoints` is irrelevant because the number of points will be determined as the number of land points the model finds within the given lat/lon range.
`readFileLand`=T indicates that the land fraction field is read from the binary file called 'grid.gra', which has no headers and contains land fraction as the first field.

With this information, JULES determines that there are 4 gridpoints within the given lat/lon range, but only 3 are land. As the 3 land points do not form a rectangle, the model grid is a vector of 3 points. As we are only modelling land points, the land grid is identical to the model grid.

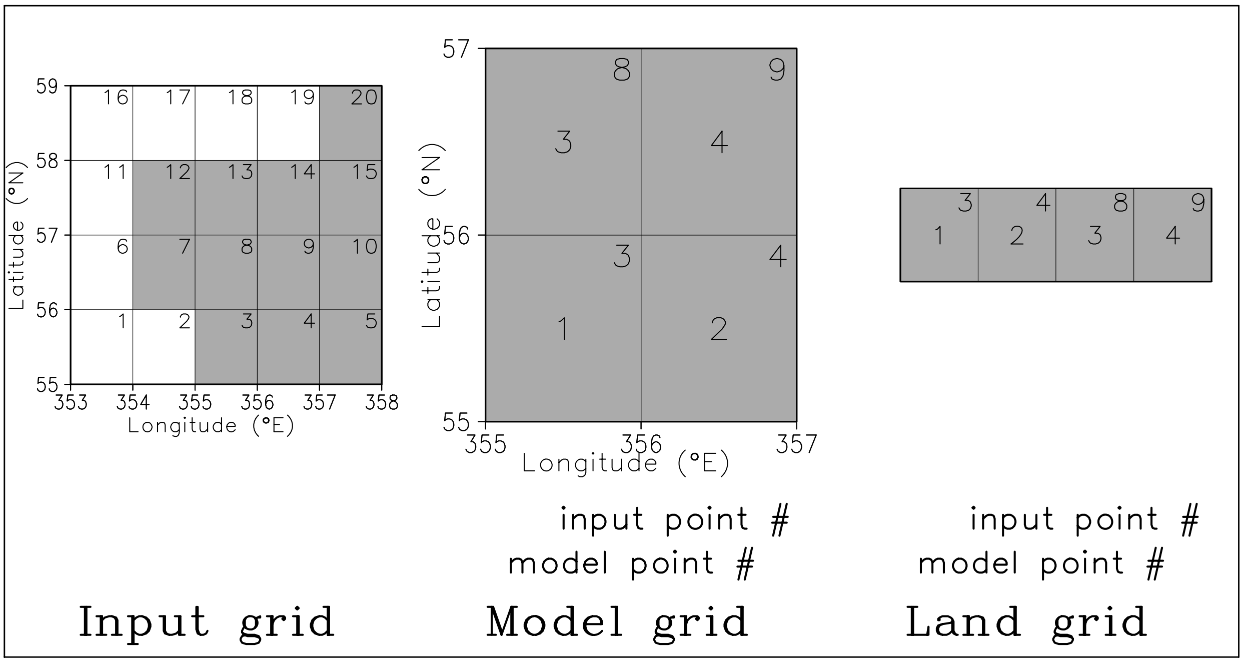

Figure 2 Example of grid selection based on longitude and latitude, taking land points only.

6.5. INIT_FRAC: Fractional coverage of land surface types

In this section, we specify the fraction of the land area in each gridbox that is covered by each of the surface types. Under certain circumstances (described below), this information may be acquired later, via another section.

```
>INIT_FRAC

readFracIC
readFile
fileFormat
filename

>ASCBIN
nheaderFile, nheaderField
fieldNum

>NC
nfracDim
fracDim(1:nfracDim)
varName

>DATA
frac(1:nxIn,1:nyIn)
```

Table 12. Description of options in the INIT_FRAC section.

Variable name	Type and permitted values	Notes
readFracIC	logical	Switch indicating location of fractional cover data. TRUE: fractional cover is provided as part of the initial condition, see INIT_FRAC in Section 6.5, and will not be read here. FALSE: fractional cover will be read here. For runs with dynamic vegetation (TRIFFID, l_trif=TRUE), the fraction cover is a prognostic variable and it must be read with the initial condition (readFracIC=TRUE).
readFile	logical	Switch controlling location of fractional cover data. Only used if readFracIC=FALSE. TRUE: read from an external file FALSE: read from the run control file.
fileFormat	character See notes in Section 5.2.	Format of data. Only used if readFile=TRUE.

`filename`	character	Name of file containing data. Only used if `readFile=TRUE`.
The following are read only if `readFile=TRUE` . Only values for the appropriate file format are read.		
`>ASCBIN`: The following are used if `fileFormat='asc', 'bin' or 'pp'`.		
`nheaderFile`	integer >=0	The number of headers at the start of the file. .See Section 5.2.
`nheaderField`	integer >=0	The number of headers before each field. See Section 5.2.
`fieldNum`	integer >=1	The number of the first field to be used from the input file (this represents the first surface type). See discussion of fields in Section 5.1.
`>NC`: The following are used if `fileFormat='nc'`.		
`nFracDim`	integer >=1	Data in SDF is held on array of `nFracDim` dimensions.
`fracDim(1:nFracDim)`	character array	Names of dimensions in file.
`varName`	character	The name of the variable containing data.
`>DATA`: The following are used if `readFile=FALSE`.		
`frac(1:nxIn,1:nyIn)`	real array >=0.0	The fractional coverage of each surface type. The fractions should sum to 1 (this is checked by the code). These values are only read if `readFile=F`, and must be located after the tag `>DATA`.

Note that all land points must be either soil points (indicated by values > 0 of the saturated soil moisture content), or land ice points (indicated by the fractional coverage of the ice surface type [if used] being one). The fractional cover of the ice surface type in each gridbox must be either zero or one – there cannot be partial coverage of ice within a gridbox.

6.5.1. Example: Reading `frac` from the run control file.

We assume `nxIn=nyIn=npoints=1`, and `ntype=9`. Only the lines relevant to this case are shown.

```
>INIT_FRAC

F            !   readFracIC
F            !   readFile

>DATA
0.55, 0.15, 0.20, 0.00, 0.05, 0.00, 0.05, 0.00    !  frac
```

`readFracIC=F` indicates that `frac` is read here, rather than as part of the initial condition.
`readFile=F` indicates that data will be read from the run control file, not from another file.
The 9 values of `frac` are positioned after the `>DATA` tag.

6.6. `INIT_DZSOIL`: Soil layer depths

```
>INIT_DZSOIL

>DATA
dzsoil(1:sm_levels)
```

Table 13. Description of options in the `INIT_DZSOIL` section.

Variable name	Type and permitted values	Notes
`dzSoil(1:sm_levels)`	real array	The soil layer depths (m), starting with the uppermost layer. Note that the soil layer depths (and hence the total soil depth) are constant across the domain. In its standard setup, JULES uses layer depths of 0.1, 0.25, 0.65 and 2.0m, giving a total depth of 3.0m.

6.7. `INIT_SOIL`: Soil hydraulic and thermal characteristics

```
>INIT_SOIL

l_vg_soil
constZ,zrev
readFile
fileFormat
filename

>ASCBIN
nheaderFile,nheaderField
>VARS
name(1),fieldNumber(1)
---- Repeated for each variable. ---
>ENDVARS

>NC
nsoilDim
soilDim(1:nsoilDim)
name(1),SDFname(1)
>VARS
---- Repeated for each variable. ---
>VARS

>INIT_SOIL2
multH,multCon

>DATA
data values
```

Table 14. Description of options in the INIT_SOIL and INIT_SOIL2 sections.

Variable name	Type and permitted values	Notes
`l_vg_soil`	logical	Switch for van Genuchten soil hydraulic model. TRUE: use van Genuchten model. FALSE: use Clapp and Hornberger model.
`constZ`	`logical`	Switch indicating if soil characteristics are to be uniform with depth at each gridbox. TRUE: soil characteristics do not vary with depth. FALSE: soil characteristics vary with depth.
`zrev`	logical	Switch indicating if input data are stored in reverse order of levels compared with JULES's expectation.

		TRUE: vertical order is reversed, with data stored in "bottom to top" order (i.e. bottom layer first) FALSE: standard vertical order, with data stored in "top to bottom" order (i.e. uppermost layer first)
`readFile`	logical	Switch controlling location of soil layer data. TRUE: read from an external file FALSE: read from the run control file.
`fileFormat`	character	Format of data file. Only used if `readFile=TRUE`.
`filename`	character	Name of file containing data. Only used if `readFile=TRUE`.
`>ASCBIN`: The following are used if `fileFormat='asc'`, `'bin'` or `'pp'`, or if `readFile=FALSE`.		
`nheaderFile`	integer >=0	The number of headers at the start of the file (not used if `readFile`=FALSE). See Section 5.2.
`nheaderField`	integer >=0	The number of headers before each field (not used if `readFile`=FALSE). See Section 5.2.
Each variable is described by a line with two values (`name` and `fieldNumber`), separated by spaces (NB no commas). The list of variables is preceeded by the tag `>VARS`, and followed by the tag `>ENDVARS`.		
`name`	character	The name of a soil hydraulic variable. These names must be chosen from the list in Table 15 below. At present all 9 variables must be provided.
`fieldNumber`	integer	The field number of the first level of data in the input file that is to be used for a variable. See discussion of fields in Section 5.1. (Note that if `readFile=FALSE`, this is interpreted slightly differently – it is the variable number, not field number.)
`>NC`: The following are used if `fileFormat='nc'`.		
`nsoilDim`	integer >=1	Data in SDF is held on array of `nsoilDim` dimensions.
`dimName(1:nsoilDim)`	character array	Names of dimensions in file.
`name`	character	See under `>ASCBIN` above.
`SDFname`	character	The name of a variable containing data, as it appears in the SDF.
Used in all cases:		
`multH`	`real`	Multiplier for matric suction at saturation (`sathh`), to convert to absolute suction (m). The input values of `sathh` are multiplied by `multH`. The suction at saturation is generally less than zero, but JULES uses the absolute value, hence it is often necessary to set `multH=-1`. Only used if `readFile=TRUE`.
`multCon`	`real`	Multiplier for saturated hydraulic conductivity (`satcon`), to convert to units of kg m^{-2} s^{-1} (which is equivalent to mm s^{-1}, assumed a density of 1000 kg m^{-3}). The input values of `satcon` are multiplied by `multCon`. Conductivity is often expressed in terms of m s^{-1}, which would require

		multCon=1000. Only used if readFile=TRUE.

>DATA:
If readFile=FALSE, data for the soil variables should be listed here in the order given in Table 15.

Table 15. List of soil parameters.

Names must be entered exactly as specified here (must be in lower case).

Name	Description
albsoil	Soil albedo. A single (averaged) waveband is used.
b	Exponent in soil hydraulic characteristics.
hcap	Dry heat capacity (J m^{-3} K^{-1})
hcon	Dry thermal conductivity (W m^{-1} K^{-1})
satcon*	Hydraulic conductivity at saturation (kg m^{-2} s^{-1})
sathh*	If l_vg_soil=TRUE (using van Genuchten model), sathh=1/α, where α is a parameter of the van Genuchten model. If l_vg_soil=FALSE (using Clapp and Hornberger model), sathh is the absolute value of the soil matric suction at saturation (m). The suction at saturation is generally less than zero, but JULES uses the absolute value.
sm_crit	Volumetric soil moisture content at the critical point (m^3 water per m^3 soil). The critical point is that at which soil moisture stress starts to restrict transpiration
sm_sat	Volumetric soil moisture content at saturation (m^3 water per m^3 soil). Note that this field is used to distinguish between soil points and land ice points. sm_sat>0 indicates a soil point.
sm_wilt	Volumetric soil moisture content at the wilting point (m^3 water per m^3 soil). The wilting point is that at which soil moisture stress completely prevents transpiration

*satcon and sathh may be adjusted through application of the multipliers multCon and multH described in this section. This only applies if data come from an external file (readFile=TRUE).

6.8. INIT_VEG_PFT: Time-invariant parameters for plant functional types

This section reads the values of parameters for each of the plant functional types (PFTs). These parameters are a function of PFT only. Parameters that also vary with time and/or location are dealt with in control file section INIT_VEG_VARY (see guide Section 6.9). Parameters that are only required if the dynamic vegetation (TRIFFID) or phenology sections are requested are read separately in control file section INIT_TRIF (see guide Section 6.12).

For many applications, the best approach may be to read the PFT parameters from the standard parameter files provided with the JULES code (readFile=TRUE, filename='PARAM/standard_pft_param.dat'), since this removes the risk that values can be changed by an accidental edit to the run control file. The description of INIT_VEG_PFT options is given in Table 16 and the list of required variables is given in Table 17.

```
>INIT_VEG_PFT

readFile
fileName
npftInFile

>DATA
var1(1),var1(2),...,var1(npft)
var2(1),var2(2),...,var2(npft)
… … data values … …
```

Table 16. Description of options in the INIT_VEG_PFT section.

Variable name	Type and permitted values	Notes
readFile	logical	Switch controlling location of data. TRUE: read from an external file FALSE: read from the run control file.
filename	character	The name of the external file containing the data. Only used if readFile=TRUE.
npftInFile	integer ≥ npft	The number of PFTs for which parameters are given in the input file.
>DATA: If readFile=FALSE, the data should be listed here (on the line after >DATA) in the order given in Table 17. Each variable should start on a new line, and npftInFile values should be given.		

Table 17. List of PFT parameters.

Each parameter has a separate value for each PFT, npftInFile values are read for each parameter. All values are of type REAL, unless stated otherwise. Parameters for the TRIFFID or phenology modules are described in Section 6.12.

HCTN24 and 30 refer to Hadley Centre technical notes 24 and 30, available from http://www.metoffice.gov.uk/research/hadleycentre/pubs/HCTN/index.html

Variable name	Description
typeName	Character. Name of each PFT. This list must include the PFTs used in this run – see pftName in section INIT_OPTS (Section 6.2). These names are for the user's convenience, and do not have any special significance within JULES.
c3	Integer. Flag indicating whether PFT is C3 type. 0 : not C3 (i.e. C4) 1 : C3
canht_ft	The height of each PFT (m), also known as the canopy height. The value read here is only used if TRIFFID is not active (l_trif=FALSE). If TRIFFID is active, canht_ft is a prognostic variable and its initial value is read as described in Section 6.16 below.
LAI	The leaf area index (LAI) of each PFT. The value read here is only used if TRIFFID is not active (l_trif=FALSE). If TRIFFID is active, LAI is a prognostic variable and its initial value is read as described in Section 6.16 below.
catch0	Minimum canopy capacity (kg m^{-2}). This is the minimum amount of water that can be held on the canopy. See HCTN30 p7.
dcatch_dlai	Rate of change of canopy capacity with LAI (kg m^{-2}). Canopy capacity is calculated as catch0 + dcatch_dlai*lai. See HCTN30 p7.
dz0v_dh	Rate of change of vegetation roughness length for momentum with height. Roughness length is calculated as dz0v_dh*canht_ft. See HCTN30 p5.
z0h_z0m	Ratio of the roughness length for heat to the roughness length for momentum. This is generally assumed to be 0.1. See HCTN30 p6. Note that this is the *ratio* of the roughness length for heat to that for momentum. It does *not* alter the roughness length for momentum, which is calculated using canht_ft and dz0v_dh (see above).
infil_f	Infiltration enhancement factor. The maximum infiltration rate defined by the soil parameters for the whole gridbox may be modified for each PFT to account for tile-dependent factors, such as marcro-pores related to vegetation roots. See HCTN30 p14 for full details.
rootd_ft	Root depth (m). An exponential distribution with depth is assumed, with e-folding depth rootd_ft. Note that this means that generally some of the roots exist at depths greater than rootd_ft. See HCTN30 Eq.32.
snowCanPFT	Flag indicating whether snow can be held under the canopy of each PFT. Only used if can_model=4 (see Section 6.2). The model of snow under the canopy is currently only suitable for coniferous trees. Acceptable values are: 0: snow cannot be held under the canopy. 1: snow can be held under the canopy.
albsnc_max	Snow-covered albedo for large leaf area index. Only used if l_spec_albedo=FALSE. See HCTN30 Eq.2
albsnc_min	Snow-covered albedo for zero leaf area index. Only used if l_spec_albedo=FALSE. See HCTN30 Eq.2.
albsnf_max	Snow-free albedo for large LAI.

	Only used `if l_spec_albedo=FALSE`. See HCTN30 Eq.1.
`kext`	Light extinction coefficient - used with Beer's Law for light absorption through tile canopies. See HCTN30 Eq.3.
`kpar`	PAR Extinction coefficient (m^2 leaf/m^2 ground)
`orient`	Flag indicating leaf angle distribution. 0 : spherical 1 : horizontal
`alpha`	Quantum efficiency (mol CO_2 per mol PAR photons).
`alnir`	Leaf reflection coefficient for NIR. HCTN30 Table 3
`alpar`	Leaf reflection coefficient for VIS. HCTN30 Table 3
`omega`	Leaf scattering coefficient for PAR.
`omnir`	Leaf scattering coefficient for NIR.
`a_wl`	Allometric coefficient relating the target woody biomass to the leaf area index (kg carbon m^{-2}).
`a_ws`	Woody biomass as a multiple of live stem biomass.
`b_wl`	Allometric exponent relating the target woody biomass to the leaf area index. This is 5/3 in HCTN24 Eq.8.
`eta_sl`	Live stemwood coefficient (kg C/m/LAI)
`G_leaf_0`	Minimum turnover rate for leaves (/360days).
`dgl_dm`	Rate of change of leaf turnover rate with moisture availability.
`dgl_dt`	Rate of change of leaf turnover rate with temperature (K^{-1}). This is 9 in HCTN24 Eq.10.
`glmin`	Minimum leaf conductance for H_2O (m s^{-1}).
`dqcrit`	Critical humidity deficit (kg H_2O / kg air). See Eqn.17 of Cox et al. (1999).
`fd`	Scale factor for dark respiration. See HCTN 24 Eq. 56.
`f0`	`CI/CA` for `DQ = 0`. See HCTN 24 Eq. 32.
`fsmc_of`	Moisture availability below which leaves are dropped.
`neff`	Scale factor relating V_{cmax} with leaf nitrogen concentration. See HCTN 24 Eq. 51.
`nl0`	Top leaf nitrogen concentration (kg N/kg C).
`nr_nl`	Ratio of root nitrogen concentration to leaf nitrogen concentration
`ns_nl`	Ratio of stem nitrogen concentration to leaf nitrogen concentration.
`r_grow`	Growth respiration fraction
`sigl`	Specific density of leaf carbon (kg C/m2 leaf).
`tleaf_of`	Temperature below which leaves are dropped (K).
`tlow`	Lower temperature for photosynthesis (deg C).
`tupp`	Upper temperature for photosynthesis (deg C).

6.9. INIT_VEG_VARY: Time-/space- varying parameters for plant functional types

This section describes prescribed characteristics of the vegetation that vary with time and/or location, in addition to varying with PFT.

```
>INIT_VEG_VARY

nvegVar
vegDataPer, vegUpdatePer
nvegFileTime, vegFilePer
vegClim
readList
fileName
vegFileDate(1),vegFileTime(1)
vegEndTime
fileFormat

>ASCBIN
nfieldFile
nheaderFile,nheaderField
noNewLineVeg
varName(1),flag(1),fieldNumber(1),interp(1),nameFile(1)
--- Repeated for each of nvegVar variables.---

>NC
nvegDim
vegDim(1:nvegDim)
varName(1),flag(1),interp(1),SDFname(1),nameFile(1)
--- Repeated for each of nvegVar variables.---
```

Table 18. Description of options in the INIT_VEG_VARY section.

Variable name	Type and permitted values	Notes
nvegVar	integer 0≤nvegVar≤ 3	The number of prescribed characteristics that vary with time and/or location. The three characteristics that may vary are vegetation height, leaf area index and root depth. If nvegVar=0, nothing more is read from this section.
vegDataPer	integer	The period (s) of time-varying data. If there are no time-varying fields, enter 0. Special cases: -1 indicates monthly data.
VegUpdatePer	integer	The period (s) between updates of time-varying fields. This must be less than or equal to the data period (vegDataPer). For example,

		`vegDataPer=86400`, `vegUpdatePer=3600`, indicates that the data are daily values and these should be updated (by interpolation) on an hourly basis. Special cases: 0: update every timestep -1: update once a month
`nvegFileTime`	integer ≥1	The number of data files available for each variable, each file holding data for different times. If all variables are held together, this is the number of data files. If variables are held in separate files, this is the number of files for any one variable.
`vegFilePer`	integer	The period (s) of the files containing the data. This must be at least as large as the period of the data (vegDataPer), and must be a multiple of the model timestep. Special cases: -1: monthly files -2: annual files
`vegClim`	logical	Switch indicating if time-varying vegetation data are to be treated as climatological, in the sense that the same data are to be used regardless of the year. TRUE: data are climatological. The year given for each file is ignored. FALSE: data are not climatological
`readList`	logical	Switch controlling how the names of the files containing the vegetation data, and the times covered by each, are read. TRUE: a list of names and times is read from another file. This is required if `nvegFileTime`>1. FALSE: a single name and file are read from the run control file. This option is only allowed if `nvegFileTime`=1 (see above).
`filename`		If `nvegFileTime=1` this is the name of the single data file (or the template). If `nvegFileTime>1`, this is the name of a file that lists the names and times of the data files. The first line of this file will be skipped (and so can be used for comments). All other lines are to be of the form `filename, startDate, ″startTime″`, where `fileName` may contain variable-name-templating (see Section 6.18). `startDate` is in the format `yyyymmdd`, and time is in the format `hh:mm:ss`.
`vegFileDate`	integer	Date of first data in vegetation file, in format `yyyymmdd`. Only used if `readList=FALSE` (otherwise read from an external file).
`vegFileTime`	character	Time of first data in vegetation file, in format `hh:mm:ss`. Only used if `readList=FALSE` (otherwise read from an external file).

vegEndTime	logical	Flag used with vegetation file templating. TRUE means that time in filename refers to the final data in the file, FALSE means the time in the filename refers to the first data in the file.
fileFormat	character See Section 5.2.	Format of vegetation data files.
The following are read only if readFile=TRUE. Only values for the appropriate file format are read.		
>ASCBIN: If fileFormat='asc', 'bin' or 'pp':		
nfieldFile	integer	Number of fields in each file.
nHeaderFile	integer >=0	The number of headers at the start of each file - see Section 5.2.
nHeaderTime	integer >=0	The number of headers at the start of each time - see Section 5.2.
nHeaderField	integer >=0	The number of headers at the start of each field - see Section 5.2.
noNewLineVeg	logical	Switch describing format of ASCII file. TRUE means that variables are arranged across one or more lines, and each variable does not necessarily start a new line. This option should be used if all the data for each time are one line of the input file (although it can also be used if the data continue onto subsequent lines). TRUE is only allowed if the fields are not functions of position (i.e. vegFlag='t', see above). FALSE means that each variable starts on a new line.
varName	character 'canht', 'lai', 'rootd'	The name of the variable. This is used to identify the variable in the code, and is set in the code. These must be entered exactly as listed, and are case-sensitive. Acceptable values: 'lai' for leaf area index 'canht' for canopy height 'rootd' for root depth
Flag	character 't', 'tx', 'x'	Flag indicating how the characteristic varies. Acceptable values: t: function of PFT and time only tx: function of PFT, time and location x: function of PFT and location only At present, all nvegVar variables must have the same value for this flag. rootd can only use flag 't' (i.e. root depth cannot vary with location).
fieldNumber	integer	The field number of the first level of data in the input file that is to be used for a variable.
interpFlag	character See Table 33.	Flag indicating how variable is to be interpolated in time.
nameFile	character	The substitution string used in the names of files that

		contain this variable. Only used if variable name templating is used (see Section 6.18).
>NC: If `fileFormat='nc'`:		
`nvegDim`	integer ≥1	Number of dimensions in SDF variable.
`vegDim(1:nvegDim)`	character array	Names of dimensions.
`varName`	character	See above under `>ASCBIN`.
`flag`	character	See above under `>ASCBIN`.
`interpFlag`	character	See above under `>ASCBIN`.
`SDFname`	character	The name of the variable as it appears in a SDF.
`nameFile`	character	See above under `>ASCBIN`.

6.9.1. Examples of `INIT_VEG_VARY`

Example 1: Time-varying Leaf Area Index.
Leaf Area Index is to vary with time (but not with position on the grid). Climatological monthly data are to be used, with values updated at the start of each day. Note that the values are always assumed to be a function of PFT. The ASCII input file is illustrated in Figure 3 and contains one month of data (for all PFTs) on a single line.

```
Month p1  p2   p3   p4   p5
1     0.5  4.0  1.0  2.0  1.0
2     0.7  4.0  1.1  2.0  1.5
3     0.9  4.2  1.5  2.0  2.0
4     2.0  4.5  2.0  2.0  2.5
---- rest of file not shown---
```

Figure 3 Schematic of an ASCII file with monthly LAI data

The relevant entries in the run control file are shown below. Only the lines in **bold** are relevant, and irrelevant sections have been omitted.

```
>INIT_VEG_VARY

1              !  nvegVar
-1,86400       !  vegDataPer, vegUpdatePer
1,1            !  nvegFileTime, vegFilePer
T              !  vegClim
F              !  readList
'lai_monthly.dat'   !  fileName
20120115,'00:00:00' !  vegFileDate(1),vegFileTime(1)
'asc'               !  fileFormat

>ASCBIN
6              !  nfieldFile
```

```
1,0           !  nheaderFile,nheaderField
T             !  noNewLineVeg
'lai',   't', 2, 'i', 'notused'  ! name, flag, field, interp, nameFile
```

nvegVar=1 indicates that we only want to vary one vegetation characteristic. vegFileDate=20120115, but since vegClim=T, the year is discarded (effectively leaving 0115=15 January), meaning that each time of data is valid on the 15th of the month. nfieldFile=6 because we have data for each of 5 PFTs, plus there is a 'timestamp' variable that will not be used (see Figure 3). The final line shows that we want to vary LAI as a function of time (and PFT) only. The LAI data start with field #2. The 'I' and vegUpdatePer=86400 indicate that the monthly data will be interpolated between the monthly values and updated once every 86400s (once a day).

6.10. `INIT_NONVEG`: Parameters for non-vegetation surface types

```
>INIT_NONVEG

readFile
fileName
nnvgInFile

>DATA
dataVar1(1),dataVar1(2),...,dataVar1(nnvgInFile)
dataVar2(1),dataVar2(2),...,dataVar2(nnvgInFile)
… … data values … …
```

Table 19. Description of options in the `INIT_NONVEG` section.

Variable name	Type and permitted values	Notes
`readFile`	logical	Switch controlling location of data. TRUE: read from an external file FALSE: read from the run control file.
`filename`	character	The name of the file to be read. Only used if readFile=TRUE. Note: For many applications, the best approach may be to read the parameters from the files provided with the JULES code (via `readFile=TRUE`), since this removes the risk that values can be changed by an accidental edit to the run control file.
`nnvgInFile`	integer ≥ nnvg	The number of non-vegetation surface types for which parameters are available in the input file.
`>DATA` The following is the list of `dataVar` parameters that must be defined for each non-PFT tile type. HCTN30 refers to Hadley Centre technical note 30, available from http://www.metoffice.gov.uk/research/hadleycentre/pubs/HCTN/index.html		
`typeName`	`character`	Name of each surface type. This list must include the non-vegetation surface types used in this run as defined in `INIT_OPTS` variable `nvgName` (see Section 6.2). Special cases: `'soil'` – this surface type must always be present. `'water'` – this is used to indicate open water, such as lakes. `'ice'` – this is used to indicate land ice, such as glaciers. Each special type must be represented by not more than one type (e.g. we cannot have two `'soil'` types).

albsnc_nvg	real	Snow-covered albedo. Only used if l_spec_albedo=FALSE. See HCTN30 Table 1
albsnf_nvg	real	Snow-free albedo. See HCTN30 Table 1 Only used if l_spec_albedo=FALSE.
catch_nvg	real	Capacity for water (kg m^{-2}). See HCTN30 p7
gs_nvg	real	Surface conductance (m s^{-1}). See HCTN30 p7 Soil conductance is modified by soil moisture according to HCTN30 Eq 35.
infil_nvg	real	Infiltration enhancement factor. The maximum infiltration rate defined by the soil parameters for the whole gridbox may be modified for each tile to account for tile-dependent factors. See HCTN30 p14
z0_nvg	real	Roughness length for momentum (m). See HCTN30 Table 4
Z0h_z0m	real	Ratio of the roughness length for heat to the roughness length for momentum. This is generally assumed to be 0.1. See HCTN30 p6. Note that this is the *ratio* of the roughness length for heat to that for momentum. It does *not* alter the roughness length for momentum, which is given by z0_nvg above.
ch_nvg	real	Heat capacity of this surface type (J K^{-1} m^{-2}). Used only if can_model is 3 or 4 (See INIT_OPTS, Section 6.2).
vf_nvg	real 0≤vf_nvg≤1	Fractional coverage of non-vegetation "canopy". Typically set to 0.0, but value of 1.0 used if tile should have a heat capacity in conjunction with can_model options 3 or 4 (See INIT_OPTS, Section 6.2)

6.11. `INIT_SNOW`: Snow parameters

```
 >INIT_SNOW

rho_snow
snow_hcap,snow_hcon
r0,rmax
snow_ggr(1:3)
amax(1:2)
dtland,kland
maskd
snowLoadLAI,snowInterceptFact,snowUnloadFact
```

Table 20. Description of options in the `INIT_SNOW` section

HCTN30 refers to Hadley Centre technical note 30, available from http://www.metoffice.gov.uk/research/hadleycentre/pubs/HCTN/index.html.

Variable name	Type and permitted values	Notes
`rho_snow`	real	Density of lying snow (kg m^{-3}).
`snow_hcap`	real	Thermal capacity of lying snow (J K^{-1} m^{-3}) Typical value=0. ·3e6
`snow_hcon`	real	Thermal conductivity of lying snow (W m^{-1} K^{-1}) See HCTN30 Eq.42 Typical value= 0·265
`r0`	real	Grain size for fresh snow (µm). See HCTN30 Eq.15. A typical value is 50·0. Only used if `l_spec_albedo=TRUE`.
`rmax`	real	Maximum snow grain size (µm). See HCTN30 p4. A typical value 2000.0 Only used if `l_spec_albedo=TRUE`.
`snow_ggr(1:3)`	real array	Snow grain area growth rates (µm ^{2} s^{-1}).. Only used if `l_spec_albedo=TRUE`. See HCTN30 Eq.16 The 3 values are for melting snow, cold fresh snow and cold aged snow respectively. Typical values are 0·6, 0·06, 0·23e6
`amax(1:2)`	real array	Maximum albedo for fresh snow. . Only used if `l_spec_albedo=TRUE`. Values 1 and 2 are for VIS and NIR wavebands respectively. Typical values=0·98, 0. ·7

`dtland`	real	Degrees Celsius below zero at which snow albedo equals cold deep snow albedo. This is 2·0 in HCTN30 Eq4. Only used if `l_spec_albedo=FALSE`.
`kland`	real	Used in snow-ageing effect on albedo. This is 0·3 in HCTN30 Eq4 (note the last term of that equation should be divided by `dtland`, i.e. `kland` as specified here includes a factor `dtland` in the denominator). Only used if `l_spec_albedo=FALSE`. Must not be zero.
`maskd`	real	Used in exponent of equation weighting snow-covered and snowfree albedo. This is 0·2 in HCTN30 Eq.5.
`snowLoadLAI`	real	Ratio of maximum canopy snow load to leaf area index (kg m^{-2}). This is 4·4 in JULES1. Only used if `can_model`=4.
`snowInterceptFact`	real	Constant in relationship between mass of intercepted snow and snowfall rate. This is 0·7 in JULES1. Only used if `can_model`=4
`snowUnloadFact`	real	Constant in relationship between canopy snow unloading and canopy snow melt rate. This is 0·4 in JULES1. Only used if `can_model`=4

6.12. `INIT_TRIF`: Parameters for the TRIFFID model

This section is used to read PFT parameters hat are only needed by the dynamic vegetation model (TRIFFID). Values are not read if TRIFFID is not selected. TRIFFID also uses many other PFT-specific variables that are also used in other parts of JULES, and are read in Section 6.8 above.

```
>INIT_TRIF

readFile
fileName
nnvgInFile

>DATA
dataVar1(1),dataVar1(2),…,dataVar1(nPft)
dataVar2(1),dataVar2(2),…,dataVar2(nPft)
… … data values … …
```

Table 21. Description of options in the `INIT_TRIF` section.

Variable name	Type and permitted values	Notes
`readFile`	logical	Switch controlling location of data. TRUE: read from an external file FALSE: read from the run control file.
`filename`	`character`	The name of the file to be read. Only used if `readFile=TRUE`.
`npftInFile`	`integer ≥npft`	The number of PFTs for which parameters are available in the input file.
>DATA If `readFile`=FALSE, the `dataVar` parameters should be listed in the order given below.		
`crop`	integer 0 or 1	Flag indicating whether the PFT is a crop. Only crop PFTs are allowed to grow in the agricultural area. 0 : not a crop 1 : a crop
`g_area`	`real`	Disturbance rate (/360days).
`g_grow`	`real`	Rate of leaf growth (/360days).
`g_root`	`real`	Turnover rate for root biomass (/360days).
`g_wood`	`real`	Turnover rate for woody biomass (/360days)
`lai_max`	`real`	Maximum LAI
`lai_min`	`real`	Minimum LAI

6.13. `INIT_AGRIC`: Fractional coverage by agriculture

If the TRIFFID vegetation model is used, the fractional area of agricultural land in each gridbox is read in this section. Otherwise, this section is not used.

```
>INIT_AGRIC

readFile
fileFormat
fileName

>ASCBIN
nheaderFile,nheaderField
fieldNum

>NC
ndim
dimName(1:ndim)
varName

# Data fields to be read from this file should appear below here.
>DATA
frac_agr(1:nxIn,1:nyIn)
```

Table 22. Description of options in the `INIT_AGRIC` section.

Variable name	Type and permitted values	Notes
`readFile`	logical	Switch controlling location of soil layer data. TRUE: read from an external file FALSE: read from the run control file.
`fileFormat`	`character`	Format of data file. Only used if `readFile=TRUE`.
`filename`	`character`	Name of file containing data. Only used if `readFile=TRUE`.
The following are read only if `readFile=TRUE`. Only values for the appropriate file format are read.		
`>ASCBIN`: The following are used if `fileFormat='asc', 'bin' or 'pp'`.		
`nheaderFile`	integer >=0	The number of headers at the start of the file,
`nheaderField`	integer >=0	The number of headers before each field.
`fieldNum`	integer >=1	The field number of the first field to be used from the input file.
`>NC`: The following are used if `fileFormat='nc'`.		

nDim	integer >=1	Data in SDF is held on array of nDim dimensions.
dimName(1:nDim)	character array	Names of dimensions in file.
SDFName[CHAR]	character	The name of the variable containing data, as it appear in the SDF.
>DATA: The following are used if readFile=FALSE.		
frac_agr(1:nxIn,1:nyIn)	real array	The fraction that is agriculture.

6.14. `INIT_MISC`: Miscellaneous surface, carbon and vegetation parameters

```
>INIT_MISC

hleaf,hwood
beta1,beta2
fwe_c3, fwe_c4
q10_leaf
kaps, q10_soil
cs_min
co2_mmr
frac_min, frac_seed
pow
```

Table 23. Description of options in the `INIT_MISC` section.

HCTN24 and 30 refer to Hadley Centre technical notes 24 and 30, available from http://www.metoffice.gov.uk/research/hadleycentre/pubs/HCTN/index.html

Variable name	Type and permitted values	Notes
`hleaf`	`real`	Specific heat capacity of leaves (J K^{-1} per kg carbon). HCTN30 p6 Typical value=5.7E4
`hwood`	`real`	Specific heat capacity of wood (J K^{-1} per kg carbon). HCTN30 p6 Typical value=1.1e4
`beta1`	`real`	Coupling coefficient for co-limitation in photosynthesis model. Cox et al. (1999), Eq.61 Typical value=0.83
`beta2`	`real`	Coupling coefficient for co-limitation in photosynthesis model. Cox et al. (1999), Eq.62 Typical value=0.93
`fwe_c3`	`real`	Constant in expression for limitation of photosynthesis by transport of products, for C3 plants. This is 0.5 in Eq.60 of Cox et al. (1999).
`fwe_c4`	`real`	Constant in expression for limitation of photosynthesis by transport of products, for C4 plants. This is 2.0×10^4 in Eq.60 of Cox et al. (1999).
`q10_leaf`	`real`	Q10 factor for plant respiration. Cox et al. (1999) Eq.66 Typical value=2.0
`kaps`	`real`	Specific soil respiration rate at 25 degC and optimum soil

		moisture (s^{-1}). HCTN24 Eq.16. Typical vale=5e-9
q10_soil	real	Q10 factor for soil respiration. HCTN24 Eq.17 Typical value=2.0
cs_min	real	Minimum allowed soil carbon (kg m^{-2}) Typical value=1.0e-6
co2_mmr	real	Concentration of atmospheric CO2, expressed as a mass mixing ratio.
frac_min	real	Minimum fraction that a PFT is allowed to cover if TRIFFID is used. Typical value=1.0e-6
frac_seed	real	Seed fraction for TRIFFID. Typical value=0.01
pow	real	Power in sigmodial function used to get competition coefficients. This is 20.0 in HCTN24 Eq.3.

6.15. `INIT_DRIVE`: Meteorological driving data

```
>INIT_DRIVE

driveDataPer
ndriveFileTime, driveFilePer
readList
fileName
driveFileDate(1),driveFileTime(1)
driveFormat

ioPrecipType,l_point_data
tForSnow
tForConv,conFrac
io_rad_type,ioWindSpeed
z1_uv, z1_tq

>ASCBIN
nfieldDriveFile
ndriveHeaderFile,ndriveHeaderTime,ndriveHeaderField
noNewLineDrive
>VARS
name(1)   fieldNumber(1)   interp(1)  nameFile(1)
name(2)   fieldNumber(2)   interp(2)  nameFile(2)
--- Repeat for each variable. ---
>ENDVARS

>NC
ndim
dimName(1:ndim)
>VARS
name(1)   SDFname(1)   nameFile(1)   interp(1)
name(2)   SDFname(2)   nameFile(2)   interp(2)
--- Repeat for each variable. ---
>ENDVARS
```

Table 24. Description of option in the `INIT_DRIVE` section.

Variable name	Type and permitted values	Notes
`driveDataPer`	integer 1 – 86400 (see notes)	The period of the driving data (s). This must be a multiple of the model timestep and must be at most 86400s (one day). 86400 must be a multiple of `driveDataPer`, so that data are read at the same times each day.
`ndriveFileTime`	integer >=1	The number of data files available for each variable, each file holding data for different times. If all variables are held together, this is the number of data files. If

		variables are held in separate files, this is the number of files for any one variable. If time templating is used (see Section 6.18), `ndriveFileTime` should be 1.
`driveFilePer`	integer	The period (s) of the files containing the data. This is only used if time templating is used (see Section 6.18). This must be at least as large as the period of the data (`driveDataPer`), and must be a multiple of the model timestep. Special cases: -1: monthly files -2: annual files
`readList`	logical	Switch controlling how the names of the files containing the driving data, and the times covered by each, are read. TRUE: names are read from another file FALSE: names are read from the run control file. This option is only allowed if `ndriveFileTime=1`.

`filename(1)`	character	If `ndriveFileTime=1` this is the name of the single data file (or the template name).
		If `ndriveFileTime>1`, this is the name of a file that lists the names and times of the data files. The first line of this file will be skipped (and so can be used for comments). All other lines are to be of the form:
		`filename, startDate,"startTime"`
		where
		`fileName` may contain variable-name-templating (see Section 6.18)
		`startDate` is in format `yyyymmdd`
		`time` is in format `hh:mm:ss`.

Starting time and date for first driving data file. Only used if `readList=FALSE` (otherwise these values are read from an external file).		
`driveFileDate`	integer	Date of first data in the driving data file, in format `yyyymmdd`.
`driveFileTime`	character	Time of day of first data in the driving data file in format `hh:mm:ss`.
`driveFormat`	character See Section 5.2.	Format of data files.
`ioPrecipType`	integer 1, 2, 3 or 4.	Flag indicating which precipitation variables are input, and how they are treated. (Note that all precipitation in JULES is considered to be either rainfall or snowfall.) 1: A single precipitation field is input. This represents the total precipitation (rainfall and snowfall). The total is partitioned between snowfall and rainfall, and large-scale and convective, using `tForSnow` and `tForConv` (see below). 2: Two precipitation fields are input, namely rainfall and snowfall. The rainfall is partitioned between large-scale and convective, using `tForConv` (see below).

		3: Three precipitation fields are input, namely large-scale rainfall, large-scale snowfall and convective rainfall. This cannot be used with `l_point_data=TRUE`. 4: Four precipitation fields are input, namely large-scale rainfall, large-scale snowfall, convective rainfall and convective snowfall. This cannot be used with `l_point_data=TRUE`. Note that this is the only option that considers convective snowfall (which is set to zero in all other cases). The concept of convective and large-scale (or dynamical) components of precipitation comes from atmospheric models, in which the precipitation from small-scale (convective) and large-scale motions is often calculated separately. If JULES is to be driven by the output from such a model, the driving data might include these components..
`l_point_data`	logical	Flag indicating if driving data are point or area-average values. This affects the treatment of precipitation input and how snow affects the albedo. TRUE: driving data are point data. Precipitation is not distributed in space (see FALSE below) and is all assumed to be "large-scale" in origin. The albedo formulation is suitable for a point. FALSE: driving data are area averages. The precipitation inputs are assumed to be exponentially distributed in space, as in UMDP25, and can include convective and large-scale components. The albedo formulation is suitable for a gridbox.
`tForSnow`	real >0	If `ioPrecipType` is 1 or 2, `tForSnow` is the near-surface air temperature (K) at or below which the precipitation is assumed to be snowfall. At higher temperatures, all the precipitation is assumed to be liquid.
`tForConv`	real >0	If `ioPrecipType` is 1 or 2, `tForConv` is the near-surface air temperature (K) at or above which the precipitation is assumed to be convective in origin. At lower temperatures, all the precipitation is assumed to be "large-scale" in origin. Also see conFrac. `tForConv` is not used if `l_point_data` is TRUE, since then there is no convective precipitation. `tForSnow` must be less than `tForConv`, implying that all solid precipitation is large-scale in origin.
`conFrac`	real >0	Convective precipitation covers the fraction `conFrac` of the gridbox.
`io_rad_type`	integer 1, 2 or 3	Flag indicating what radiation fluxes are input. 1: Downward fluxes of short- and longwave radiation are input. Normally this is the preferred option. 2: Downward shortwave and net (all wavelengths)

		downward radiation are input. The modelled albedo and surface temperature are used to calculate the downward longwave flux. 3: Net downward fluxes of short- and longwave radiation are input. The modelled albedo and surface temperature are used to calculate the downward fluxes of shortwave and longwave radiation.
`iowindSpeed`	logical	Switch indicating how wind data are input. TRUE: the wind speed is input FALSE: the two components of the horizontal wind (e.g. the southerly and westerly components) are input.
`z1_uv`	real >0.0	The height (m) at which the wind data are valid. This height is relative to the zero-plane *not* the ground.
`z1_tq`	real >0.0	The height (m) at which the temperature and humidity data are valid. This height is relative to the zero-plane *not* the ground.
`>ASCBIN: If fileFormatLatLon='asc', 'bin' or 'pp':`		
`nfieldFile`	integer	Number of fields in each file.
`nHeaderFile`	integer >=0	The number of headers at the start of each file - see Section 5.2.
`nHeaderTime`	integer >=0	The number of headers at the start of each time - see Section 5.2.
`nHeaderField`	integer >=0	The number of headers at the start of field - see Section 5.2.
`noNewLineDrive`	logical	Switch describing format of an ASCII data file. TRUE: variables are arranged across one or more lines, and each variable does not necessarily start a new line. This option should be used if all the driving data for each time are one line of the input file (although it can also be used if the data are continued onto subsequent lines). FALSE: each variable starts on a new line. Only used if there is only one point in the input grid (and hence only one point in the model grid) and driving data are in ASCII files.
`name`	character	The name of the variable. This is used to identify the variable in the code, and is set in the code. Acceptable values are shown in Table 25. These must be entered exactly as listed in the table, and are case-sensitive.
`fieldNumber`	integer >=1	The field number in the file that holds data for this variable. See discussion of fields in Section 5.
`interpFlag`	character See Table 33.	Flag indicating how variable is to be interpolated in time
`varNameFile`	character	The substitution string used in the names of files that contain this variable. Only used if variable name templating is used in file names.
>NC: If `fileFormat='nc':`		

ndim	integer >=1	Number of dimensions in the netCDF file.
dimName(ndim)	character array	Name of each of the dimensions in the netCDF file.
name	character	See above under >ASCBIN.
SDFName	character	The name of the variable as used in a SDF. See discussion of SDF in Section 5.2.2.
varNameFile	character	See above under >ASCBIN.
interpFlag	character See Table 33.	See above under >ASCBIN.

The meteorological variables required by a run of JULES are determined by the choice of flags such as ioPrecipType. The variables that are listed must then match this expectation.

Table 25. Names of meteorological driving variables.

Name	Description	Comments
lw_down	Downward longwave radiation (W m^{-2}).	Used with rad_type=1.
lw_net	Net downward longwave radiation (W m^{-2}).	Used with rad_type=2.
sw_down	Downward shortwave radiation (W m^{-2}).	Used with rad_type=1.
sw_net	Net downward shortwave radiation (W m^{-2}).	Used with rad_type=2.
rad_net	Net (all wavelength) downward radiation (W m^{-2}) .	Used with rad_type=3.
precip	Precipitation rate (kg m^{-2} s^{-1}).	Used with precipType=1.
precipCR	Convective rainfall rate (kg m^{-2} s^{-1}).	Used with precipType=3 and 4.
precipCS	Convective snowfall rate (kg m^{-2} s^{-1}).	Used with precipType=4.
precipLR	Large-scale rainfall rate (kg m^{-2} s^{-1}).	Used with precipType=3 and 4.
precipLS	Large-scale snowfall rate (kg m^{-2} s^{-1}).	Used with precipType=3 and 4.
precipTR	Rainfall rate (kg m^{-2} s^{-1})	Used with precipType=2.
precipTS	Snowfall rate (kg m^{-2} s-1).	Used with precipType=2.
pstar	Air pressure (Pa)	
q	Specific humidity (kg kg^{-1})	
t	Air temperature (K)	
u	Zonal component of the wind (m s^{-1}).	Used with windSpeed=FALSE.
v	Meridional component of the wind (m s^{-1}).	Used with windSpeed=FALSE.
wind	(Total) wind speed (m s^{-1}).	Used with windSpeed=TRUE.

6.15.1. Examples of specifying driving data

Example 1: single point driving data
In this example, we consider a case with one point in the input file, and all driving data for each time held on a single line of an ASCII input file. The input file is illustrated in Figure 4. The

relevant entries in the run control file are shown below. Only the lines in **bold** are relevant, and irrelevant sections have been omitted.

```
>INIT_DRIVE
3600.0                 !  driveDataPer
1,-9                   !  ndriveFileTime, driveFilePer
F                      !  readList
‘data1.dat’            !  fileName
19970101,’00:00:00’ !  driveFileDate(1),driveFileTime(1)
‘asc’                  !  driveFormat

2,T                    !  ioPrecipType,l_point_data
275.0                  !  tForSnow
375.0,0.2              !  tForConv,conFrac
1,T                    !  io_rad_type,ioWindSpeed
F                      !  ioTotalRunoff
10.0,10.0              !  z1_uv, z1_tq

>ASCBIN
10         !  nfieldDriveFile
0,0.0      !  ndriveHeaderFile,ndriveHeaderTime,ndriveHeaderField
T          !  noNewLineDrive
>VARS
pstar       9  nf  psfc   !  name,field,flag,name
t           6  nf  t
q          10  nf  q
wind        8  nf  u
lw_down     3  nf  lw
sw_down     2  nf  sw
precipTR    4  nf  liqp
precipTS    5  nf  solp
>ENDVARS
```

ndriveFileTime=1 indicates that all data are in one file.
readList=FALSE indicates that the name of the file is read from the run control file (not from a separate file).
noNewLineDrive=TRUE shows that each variable is not on a new line (in fact all variables are on one line).
The entries following >VAR indicate where each variable lies in the input file. Note that we can skip the unrequired ‘time’ and ‘obs1’ fields in Figure 4.

```
Time   solar  long   rain   snow   temp          obs1   wind   press          humid
 1     3.3    187.8  0.0    0.0    259.10        83.0   3.610  102400.5       1.351E-03
 2     89.5   185.8  0.0    0.0    259.45        24.1   3.140  102401.9       1.357E-03
 3     142.3  186.4  0.0    0.0    259.85        56.9   2.890  102401.0       1.369E-032
----- data for later times ----
```

Figure 4. Lines of an example file of meteorological driving data in ASCII format.

Example 2: Driving data from binary files, one variable per file.
The relevant entries in the run control file are shown below. Only the lines in **bold** are relevant and irrelevant sections have been omitted.

```
>INIT_DRIVE

3600.0                 !  driveDataPer
162,-9                 !  ndriveFileTime, driveFilePer
T                      !  readList
'file_list.txt'          !  fileName
19820701,'03:00:00' !  driveFileDate(1),driveFileTime(1)
'bin'                  !  driveFormat

2,F                    !  ioPrecipType,l_point_data
275.0                  !  tForSnow
298.2,0.3              !  tForConv,conFrac
1,F                    !  io_rad_type,ioWindSpeed
F                      !  ioTotalRunoff
10.0,10.0              !  z1_uv, z1_tq

>ASCBIN
1          !  nfieldDriveFile
0,0.0      !  ndriveHeaderFile,ndriveHeaderTime,ndriveHeaderField
T          !  noNewLineDrive
>VARS
pstar       1  nf  psfc    !  name,field,flag,name
t           1  nf  temp
q           1  nf  humid
u           1  nf  uwind
v           1  nf  vwind
lw_down     1  nf  long
sw_down     1  nf  solar
precipTR    1  nf  liqp
precipTS    1  nf  solp
>ENDVARS
```

`ndriveFileTime`=162 indicates the number of files (for each variable).
`readList`=TRUE indicates that the names and times of each file are read from the file `'file_list.txt'`. The first few lines of this file are shown in Figure 5.

```
# List of meteorological data files. Columns are:
# file name, start date (yyyymmdd), start time (hh:mm:ss).
'met_data/%vv_data/%vv198207.dat', 19820701, '03:00:00'
'met_data/%vv_data/%vv198208.dat', 19820801, '03:00:00'
'met_data/%vv_data/%vv198209.dat', 19820901, '03:00:00'
------ rest of file not shown -----
```

Figure 5. Example list of driving data files using file name templating.

The presence of '`%vv`' in each file name shows that we are using variable name templating (see Section 6.18). The dates show that we in fact have monthly files (but note that we cannot use time templating for these files because the start time of 03H does not conform to the requirements described in Table 31). Furthermore, files for each variable are stored in separate directories. For example, skipping ahead to after `>VARS`, we see that the humidity variable is held in files such as `'met_data/humid_data/humid198207.dat'`, while the surface pressure is held in the likes of `'met_data/psfc_data/psfc198207.dat'`.

The `ioPrecipType` value of 2 shows that we read in two components of precipitation: total solid and total liquid. The liquid is considered to be convective precipitation when the temperature is above `tForConv`, which here has a value of 298.2 K.

`nfieldDriveFile=`1 shows that each data file contains a single field, which is consistent with the `field` number shown for each variable (all 1).

6.16. `INIT_IC`: Specification of the initial state

The values of all prognostic variables must be set at the start of a run. This initial state, or initial condition, can be read from a “dump” from an earlier run of the model, or may be read from any other file. Another option is to prescribe a simple or idealised initial state, and this may be done via the run control file. It is also possible to set some fields using values from a file (e.g. a dump) but to set others using idealised values from the run control file (that is, effectively to override the values in the external file).

```
>INIT_IC

readFile
fileFormat (quoted)
fileName (quoted)
zrev

>ASCBIN
nheaderFile, nheaderField
>VARS
varName(1)   varFlag(1)   constVal(1)
varName(2)   varFlag(2)   constVal(2)
--- Repeat for each variable. ---
>ENDVARS

>NC
nDim
‘dimName(1)’,’dimName(2)’,...,’dimName(ndim)’
>VARS
varName(1)   varFlag(1)   constVal(1)   SDFname(1)
varName(2)   varFlag(2)   constVal(2)   SDFname(2)
--- Repeat for each variable. ---
>ENDVARS

# Data fields to be read from this file should appear below here.
>DATA
```

Table 26. Description of options for `INIT_IC` section.

Variable name	Type and permitted values	Notes
`readFile`	logical	Switch controlling location of initial state data. TRUE: read from an external file (including a model dump) FALSE: read from the run control file.
`fileFormat`	character	Format of data. Only used if `readFile=TRUE`. A model dump is indicated by `fileFormat=’dump’`, other valid formats are described in Section 5.2.

`filename`	character	Name of file containing data. Only used if `readFile=TRUE`.
`zrev`	logical	Switch indicating if soil data are stored in reverse order of levels. TRUE: vertical order is reversed, with data stored in "bottom to top" order (i.e. bottom layer first) FALSE: standard vertical order, with data stored in "top to bottom" order (i.e. uppermost layer first)
`>ASCBIN: The following are used if fileFormat='asc', 'bin', 'dump' or 'pp', or if readFile=FALSE.`		
`nheaderFile`	integer	The number of headers at the start of the file. See Section 5.2
`nheaderField`	integer	The number of headers at the start of a field. See Section 5.2
`varName`	character See Table 3.	The name of the variable.
`varFlag`	integer ≥-1	Flag indicating how the variable is initialised. Acceptable values: >0: The field number in the file that holds data for this variable. See discussion of fields in Section 5.1. If a dump file is being read, any integer ≥0 is accepted and then effectively ignored – this indicates that the field is to be taken from the dump and the exact field number is not required. -1: The field will be set to the value `constVal` (see below) at all points. This option can be used to specify an idealised initial condition.
`constVal`	real	The value to be used at all points. Only used if `flag=-1`.
>NC: The following are used if `fileFormat='nc'`.		
`ndim`	integer ≥1	Number of dimensions in the netCDF file.
`dimName(1:ndim)`	character array	List of names of dimensions in the netCDF file.
`varName`	character See Table 5.	The name of the variable.
`varFlag`	integer ≥-1	Flag indicating how the variable is initialised. Acceptable values: >0: Default (effectively is ignored). -1: The field will be set to the value `constVal` (see below) at all points. This option can be used to specify an idealised initial condition.
`constVal`	real	See under >ASCBIN above.
`SDFvarName`	character	The name of the netCDF variable that is to be used.

If further initial data are to be read from the run control file (`readFile=TRUE` and `fileFormat≠'dump'`), these should now appear in the file, in the order indicated by the value of flag for each variable (see above). For example, if `tstar` is given a value of `flag=1`, and `cs` has `flag=2`, data for `tstar` and `cs` should then be listed, with each variable starting on a separate line.

Some of these variables may not be required for a particular run, depending on the model configuration. The size of each variable is defined in terms of four variables: `land_pts`, the number of gridboxes that contain any land; `sm_levels`, the number of soil layers; `ntiles`, the number of tiles at each gridbox; `ntype`, the number of surface types (see Section 6.2 for description of these last three variables).

Table 27. JULES variables that define the model state (prognostic variables).

Name	Shape	Description.	Notes
`sthuf`	`(land_pts, sm_levels)`	Soil wetness for each soil layer. This is the mass of soil water (liquid and frozen), expressed as a fraction of the water content at saturation.	Either `sthuf` or its components `sthu` and `sthf` are always required.
`sthf`	`(land_pts, sm_levels)`	Frozen soil wetness for each soil layer. This is the mass of frozen water, expressed as a fraction of the water content at saturation. Note that the partitioning of water between liquid and solid fractions may be altered during initialisation. The procedure conserves the total water content, and uses the soil temperature (`t_soil`) to partition between the phases.	Either `sthuf` or its components `sthu` and `sthf` are always required.
`sthu`	`(land_pts, sm_levels)`	Unfrozen soil wetness for each soil layer. This is the mass of unfrozen water, expressed as a fraction of the water content at saturation. See notes for `sthf` above.	Either `sthuf` or its components `sthu` and `sthf` are always required.
`canopy`	`(land_pts, ntiles)`	Amount of intercepted water that is held on each tile (kg m^{-2}).	Always required.
`snow_tile`	`(land_pts, ntiles)`	Amount of snow (on the vegetation canopy) of each tile (kg m^{-2}).	Always required. Note that if snow is allowed below the canopy (`can_model=4` selected in `INIT_OPTS`, see Section 6.2), the total amount of snow held on a vegetation tile is the sum of '`snow_tile`' and '`snow_grnd`' (see below). For `can_model`≠4,

			‘snow_tile’ is the total snow on the tile.
tstar_tile	(land_pts, ntiles)	Temperature of each tile (K). This is the surface or skin temperature.	Always required.
t_soil	(land_pts, sm_levels)	Temperature of each soil layer (K).	Always required.
cs	(land_pts)	Soil carbon (kg m^{-2})	Always required. This is a prognostic variable only if TRIFFID is selected.
gs	(land_pts)	Stomatal conductance for water vapour (m s^{-1}).	Always required. This is used to start the iterative calculation of gs for the first timestep only.
frac	(land_pts, ntype)	The fraction of land area of each gridbox that is covered by each surface type.	Always required, but can be read at INIT_FRAC. This variable has to be set either in this section of the run control file, or in the section tagged INIT_FRAC (see Section 6.5). If TRIFFID is being used (see Section 6.2), frac must be set here, as part of the initial condition (e.g. from a model dump). If TRIFFID is not being used (i.e. the fraction of each type is static), the fraction may be set here, as part of the initial condition, or in INIT_FRAC. The switch readFracIC, described in that section, is important in this case.
rgrain	(land_pts, ntiles)	Snow grain size (μm) on each tile.	Only used if l_spec_albedo = TRUE.
snow_grnd	(land_pts, ntiles)	The amount of snow on the ground, beneath the canopy (kg m^{-2}), on each tile.	Only used if can_model=4. A value should be given for all tiles, but it is only updated for tiles that refer to PFTs that have snowCanPFT=1 (see Section 6.8).
lai	(land_pts, npft)	Leaf area index of each PFT.	Only set here if TRIFFID is switched on in INIT_OPTS (see Section 6.2). If TRIFFID is not used, LAI is not a prognostic variable and it is initialised in either INIT_VEG_PFT or INIT_VEG_VARY.
canht_ft	(land_pts, npft)	Height (m) of each PFT.	Only set here if TRIFFID is used – see comments for lai above.

6.16.1. Examples of specification of initial state

Example 1: A single point, state from the run control file

In this example, we consider a run at a single point and read all data from the run control file. The relevant entries in the run control file are shown below. Only the lines in **bold** are relevant and irrelevant sections have been omitted.

```
>INIT_IC

F                               !    readFile
'asc'                           !    fileFormat (quoted)
'a0001_dump.19970105'           !    fileName (quoted)
F        !  zrev

>ASCBIN
0,0                       !  nheaderFile,nheaderField
>VARS
sthuf        1     0.9   !  varName,varFlag, constVal
canopy       2     0.0
snow_tile    3     0.0   !  Note that none of these "constVal"
tstar_tile   4   275.0   !  values are used in this case.
t_soil       5   278.0   !  Instead, values are listed after >DATA.
cs           6     0.0
gs           7     0.0
>ENDVARS

# Data fields to be read from this file should appear below here.
>DATA
 0.749, 0.743, 0.754, 0.759  !  sthuf
9*0.0                        !  canopy
9*0.0                        !  snow_tile
9*276.78                     !  tstar_tile
276.78,277.46,278.99,282.48  !  t_soil
12.100                       !  cs
0.0                          !  gs
```

`readFile`=FALSE indicates that all data will be read from the run control file; no other file is involved. In this case, we use the `>ASCBIN` section to describe the data.

The seven variables that are required to initialise this particular run are then listed. The second entry in each line gives the position in the input data for each field. Since all the data are to be read from the run control file, which is easily edited, it is easiest to list these variables in the order in which the data will be presented (i.e. field numbers should be 1, 2, 3,…). In this example, all the field numbers are >0, indicating that the data will be read from the `>DATA` section (and that the `constVal` entries will be ignored).

Note that data for soil variables are presented in the order "top to bottom", i.e. surface layer first.

Example 2: Initial state specified as a mixture of spatial fields and constant values

In this example, we consider a run at a single point and read all data from the run control file. The relevant entries in the run control file are shown below. Only the lines in **bold** are relevant and irrelevant sections have been omitted.

```
>INIT_IC

T                                !    readFile
'bin'                            !    fileFormat (quoted)
'a001_initial_state.gra'         !    fileName (quoted)
T                                !    zrev

>ASCBIN
0,0                        !  nheaderFile,nheaderField
>VARS
sthuf        7     0.9   !  varName,varFlag, constVal
canopy      -1     0.0
snow_tile   -1     0.0
tstar_tile  -1   275.0
t_soil      -1   278.0
cs          -1    10.0
gs          -1     0.0
>ENDVARS
```

readFile=TRUE indicates that the binary file "a001_initial_state.gra" will be used to set the initial state (for some variables).
The seven variables that are required to initialise this particular run are then listed. The second entry in each line gives the position in the input data for each field. For most variables, the value -1 indicates that the field is to be initialised as spatially constant using the value given under constVal. For example, the temperature in each soil layer (t_soil) will be set to 278K at all locations in the model grid. For soil wetness (sthuf), the field number is given as 7 – meaning that soil wetness will be set using the data starting at field 7 in the named input file. Since zrev=TRUE, these data are stored in the file in "non-standard" order (i.e. bottom to top), so that field 7 is the deepest layer (and, assuming 4 soil layers, field 10 will be used for the uppermost layer).

6.17. `INIT_OUT`: Specification of output from the model

JULES separates output into one or more output 'profiles' or streams. Within each profile, all variables selected for output are written to the same file, with the same frequency, although the time-processing can differ between variables (e.g. instantaneous values and time-averages can appear in the same profile). Each profile can be considered as a separate data stream. By using more than one profile the user can, for example,

- Output one set of variables to one file, and other variables to another file
- Write instantaneous values to one file, and time-averaged values to another.
- Write low-frequency output from the entire model grid to one file, and high-frequency output from a subset of points to another file.
- Write low-frequency output throughout the run to one file, and high-frequency output from a smaller part of the run (e.g. a "Special Observation period") to another file.

This flexibility comes at the expense of having to set several values in the run control file. However, default values allow the user to select certain configurations relatively easily.

The first values in this section of the run control file concern general details of the output, such as the file format, that apply to all output profiles. This is followed by a separate section for each output profile, describing the variables, the grid and time sampling for that profile.

6.17.1. `INIT_OUT`: general values

This section starts with the tag `>INIT_OUT`

```
>INIT_OUT

run_id
outFormat
outDir
outStatus
yrevOut,zrevOut
numMonth
useTemplate
nout
undefOut
zsmc,zst
outEndian

dumpFreq,dumpStatus
```

Table 28. Description of options in the `INIT_OUT` section.

Variable name	Type and permitted values	Notes
`runID`	character*10	A name or identifier for the run. This is used to name output files and any model dumps.
`outFormat`	character	The format for output files. Acceptable values are: ‘`asc`’: ASCII files ‘`bin`’: flat binary files ‘`nc`’: netCDF files
`outDir`	character*150	The directory used for output files. This can be an absolute or relative path. Enter “.” to write output to the directory from which JULES is run.
`outStatus`	character ‘new’ or ‘replace’	The status used when opening files. This is the value given to the FORTRAN “status” argument of an OPEN statement [e.g. `open(1,status='new')`].Acceptable values are: ‘`new`’: file must not already exist. If the code tries to create a file with the same name as an existing file, the run will terminate. ‘`replace`’: If the file exists, delete it and replace with a new version.
`yrevOut`	logical	TRUE: reverse the order of the rows in the output, so that these are written in “North to South” order. FALSE: use the default “South to North” order, with the southernmost row of data being the first in the file.
`zrevOut`	logical	Switch indicating if soil layer data are to be output in reverse order of levels compared with JULES’s default. TRUE: reverse the order of the vertical levels in the output, so that these are written in “bottom to top” order (i.e. bottom layer first). FALSE: use the default “top to bottom” order (i.e. top layer first). This flag only applies to variables on the soil levels.
`numMonth`	logical	Switch controlling the date format used in file names. TRUE: months are represented by the numbers 1 to 12. FALSE: months are represented by 3-character strings (jan, feb, mar,…)
`useTemplate`	logical	This relates to GrADS gridded files (generated by `outFormat='bin'`). Switch to activate the writing of template ‘`.ctl`’ files. A template ctl file allows GrADS to access several data files via one ctl file. TRUE: all suitable ctl files will use the template option. FALSE: generate a separate ctl file for each data file.

		Note: A template ctl file will not work be able to describe the data if there are any missing times at the start of a file – this is a limitation of the current JULES code, rather than GrADS. For example, if daily data are to be written to monthly files, with a template ctl, but the run starts midway through the month, JULES will only write output data for the latter part of the month. GrADS will look for data for all days in the month, but not be able to find them, so the user will not be able to plot the first month.
`nOut`	integer	The number of output profiles. Each profile generates a separate stream of data, as explained above.
`undefOut`	real	The value written to output files to represent "missing" or "undefined" data.
`zsmc`	real	If a depth-averaged soil moisture diagnostic is requested, the average is calculated from the surface to this depth (m).
`zst`	real	If a depth-averaged soil temperature diagnostic is requested, the average is calculated from the surface to this depth (m).
`outEndian`	`character` `'little_endian'` or `'big_endian'`	Only used for GrADS output files (`outFormat='bin'`), this describes the byte ordering of the computers on which JULES is run. It is included in the 'options' line of GrADS ctl files. Acceptable values are: '`little_endian`' – for little endian computers (e.g. PCs) '`big_endian`' – for big endian computers (e.g. Suns)
`dumpFreq`	`integer` `0 to 4`	Flag indicating how often the model state is to be 'dumped' (written to a file). Acceptable values are: 0: no dumps are written 1: only the final state of the model (at the end of the integration) is dumped 2: dump initial and final model states 3: as 2 but also write a dump at the end of the spin-up phase 4: as 3 but also write a dump at the end of each calendar year. A model dump captures the state of the model at a given point in the integration. If a final dump is saved, the integration can later be extended by starting another run from this final dump. For long integrations, or large domains, it is recommended that dumps are saved for every year, so that in the event of any trouble such as a model crash, the integration can be completed without having to start again from the initial state. NB A run that is carried out in several steps, each starting from

		the model dump for the previous step, will generally not evolve identically to a single run that proceeds without the intermediate dumps. This is due, in part, to a loss of precision when the model state is written to the dump file.
`dumpStatus`	`Character` `'new'` or `'replace'`	The file status used when writing a model dump. Acceptable values are: `'new'` – if a file with the same name already exists, the run will terminate. `'replace'` – if a file with the same name already exists, it will be overwritten.

6.17.2. NEWPROF: details of each output profile

Each of the `nout` output profiles requires a section that describes that profile, such as the times when output is to be generated, which points are to be output, which variables are to be output, and more. The size of a regular latitude/longitude gridbox (input as `regDlat`, `regDlon` in control file – see Section 6.4.3) is also used as the size of a gridbox in the output.

```
>NEWPROF

outName
outPer,outFilePer
outSamPer
outDate(1),outTime(1)
outDate(2),outTime(2)

pointsFlag(1:2)
outAreaLL
outRangeX(1:2),outRangeY(1:2
outCompress,outLLorder

readFile
fileName

pointsOut
mapOut(1:pointsOut,1)
mapOut(1:pointsOut,2)

>GRID
outGridNx,outGridNy

>VARS
flag name useName
--repeat for each output variable --
>ENDVARS
```

Table 29. Description of options for each output profile.

Variable name	Type and permitted values	Notes
outName	character	The name of this output profile. This is used in file names and should be specified, even if there is only one profile. The names might reflect the variables in the file (e.g. 'soil'), or if several profiles are used they could be given names such as 'p1','p2'.
outPer[5]	integer	The period for output (seconds). This must be a multiple of the timestep length (except for the special cases <0 given below). It must not exceed 30 days (2592000 seconds), except for the special cases. Special cases: 0: generate output every timestep. -1: monthly period -2: annual period (calendar years)
outFilePer[5]	integer	The period for output files (seconds), i.e. the time interval within which all output goes to the same file. This must not exceed 30 days (2592000 seconds), except for the special cases given below. The file period must be consistent with the output period (e.g. we can't have daily files for monthly output). Output may be generated for only part of a run (see outDateStart below), and outFilePer controls how the data are stored during that part of the run when the output is "active". Special cases: 0: output is every timestep, and a new file is created every timestep -1: monthly files (all output for a month goes to the same file) -2: annual files (calendar years) -7: all output goes to one file, but each cycle of spin up creates a separate file -8: all output goes to one file, but all output during spin up goes to a separate file -9: all output (for all times) from this profile goes to one file
outSamPer[5]	integer	The sampling period (seconds) for time-averages and accumulations. This must be a factor of the output period (outPer). Special case: 0 means sample every timestep. In some cases, sampling every timestep adds a considerable computational burden, and acceptable output can be achieved by sampling less frequently. For example, with a large domain, many output diagnostics, and a timestep of 30 minutes, a monthly average would be calculated from several hundred values if every timestep was used. For variables that evolve relatively slowly, an acceptable monthly average might be obtained by sampling only every 12 hours.
outDateStart	integer	Date in format yyyymmdd. Output from this profile is first

[5] Many variables that are input in terms of seconds (such as outPer and outFilePer) are converted within the code to a number of model timesteps.

		generated at the date and time indicated by `outDateStart` and `outTimeStart`. These must be within the "main run", except for the special cases noted below. Note that output is only generated at the end of a timestep, except for the special cases noted below. Special cases for `outDateStart`: 0: output all times through the run, including any spin-up[6] -1: output at all times after spin-up is complete -2: output only at the start of the first timestep of the run (used to output the initial state only). Note that, at present, the only time at which output can be generated at the *start* of a timestep is at the start of the run, when `outDateStart=-2` will output the initial state. Thus the only way in which the initial state can be output is to have an output profile with `outDateStart=-2`. All output at later times then has to be generated via another output profile. (This is a slight oversimplification – see footnote 6!)
`outTimeStart`	character *8	Time of day (in format hh:mm:ss) at which output begins. Not used if `outDateStart` is one of the special cases.
`outDateEnd`	integer	Date on which output ends. Not used if `outDateStart` is one of the special cases.
`outTimeEnd`	character *8	Time of day at which output ends. Not used if `outDateStart` is one of the special cases.
`pointsFlag(1)`	integer 0, 1, 2	Flag indicating how the points to be output are selected. 0 = all points in the model grid will be output 1 = points in a rectangular subsection will be output. 2 = the points to be output will be listed individually
`pointsFlag(2)`	integer 0 to 4	Flag indicating how the locations in the output grid of output points will be calculated. 0 = the output grid will be the model grid 1 = the output grid will be the rectangular subsection specified via `pointsFlag(1)=1`. This option can only be used in conjunction with `pointsFlag(1)=1`. 2 = the location of each output point will be listed individually. This option can only be used in conjunction with `pointsFlag(1)=2`. 3 = the output grid will be the smallest rectangle that contains all the output points. This option requires that the model grid is rectilinear (or is a subset of such a grid). 4 = the output grid will be the same as the input grid. Depending upon the shapes of the input and model grids, it may be possible to produce the same output grid via different combinations of the values of `pointsFlag`. Similarly, certain combinations will be less useful for particular grids.
`outAreaLatLon`	logical	Switch indicating how to interpret the coordinates `outRangeX` and

[6] Under some circumstances, `outDateStart=0` will also output the initial state of the model. These circumstances are that the period of the output equals the timestep (i.e. information for every timestep) and that all output goes to a single file (`outFilePer=-9`). The timestamp information included with the output allows the user to determine whether this initial state has been output.

		`outRangeY`. Only used if `pointsFlag(1)=1.` TRUE: co-ordinates are longitude and latitude. FALSE: co-ordinates are x and y indices (column and row numbers).
`outRangeX(1:2)`	real array	x-coordinates of the sub-area to be output. Depending on `outAreaLatLon`, these are longitudes (in range -180 to 360º) or column numbers. Only used if `pointsFlag(1)=1`. Column numbers are those in the INPUT grid. If values are column numbers, the code uses the nearest integer to the input value.
`outRangeY(1:2)`	real array	As `outRangeX`, expect in latitudinal (y) direction.
`outCompress`	logical	Switch indicating if output data are to be "compressed" so that only model points are output. TRUE: Only output model points. Also output the mapping between the model points and the output grid (e.g. how to scatter the output points across a larger grid). The mapping is output in a form suitable for use with GrADS' pdef. FALSE: If the output grid is larger than the number of points to be output, the grid is filled with "missing data" or padding values. See Section 6.17.3 for further discussion of output compression. If the output grid is the same size as the number of points to be output (so no compression is possible), `outCompress=TRUE` may still cause output to differ in format from `outCompress=FALSE` (the points may be written in a different order), so `outCompress` should always be set to FALSE unless needed otherwise. Note that if `outCompress=TRUE`, then `yrevOut` is ignored for the profile (it becomes irrelevant).
`outLLorder`	logical	Switch indicating the coordinate system to be used to determine the locations of the output points in the output grid. Only used if `pointsFlag(2)=1` or `3.` TRUE: use the latitude and longitude of each point to determine its location in the output grid. FALSE: use the row and column number in the INPUT grid to determine where each point goes in output grid. This option is particularly useful if the input grid is rectilinear but is not regular in latitude and longitude (e.g. it could be a rotated grid). The output can then be placed on the same rectilinear grid.
`readFile`	logical	Switch controlling location of output mapping. Only used if `pointsFlag(1)=2` (i.e. a mapping is to be input). TRUE: read from an external file FALSE: read from the run control file.

`filename`	character	The name of the file that contains output mapping. Only used if `readFile=TRUE`.
`pointsOut`	integer 1 to size of grid	The number of points to be output. This is only used if `pointsFlag(1)=2`.
`mapOut(1:pointsOut,1)`	integer array	A list of the points that are to be output. The list gives the locations (point numbers) in the INPUT grid (which need not be the same as the model grid). Only used if `pointsFlag(1)=2`.
`mapOut(1:pointsOut,2`	integer array	A list giving the destination (location in output grid) for each output point. The list gives the point number in the output grid. Only used if `pointsFlag(2)=2`
`outGridNx`	integer	Number of columns in the output grid. This is the full, uncompressed output grid. If compression is applied, the actual output may be smaller, but can be scattered across a grid with this number of columns. Only used if `pointsFlag(2)=2`, in which case the user specifies all aspects of the output grid and mappings. Otherwise the size of the output grid is calculated by the model.
`outGridNy`	integer	As `outGridNx`, but number of rows.

`>VARS`
`A list of variables to be output is provided between the tags >VARS and >ENDVARS.`		

`flag`	character*1 S, M or A	Flag indicating type of processing. Acceptable values are, `S`: Instantaneous or snapshot value. `M`: Time mean value. `A`: Accumulation over time. For time averaged variables, the period over which each time average is calculated is given by `outPer`. For time-accumulation variables, `outPer` gives the period for output of an updated accumulation (i.e., how often the value if reported). For both time averages and accumulations, the sampling frequency is set via `outSamPer`. NB A time-accumulation is initialised at the start of a run (actually at the start of each section of a run so that it is reinitialised after any spin up is completed – see Section 6.3.3) and thereafter accumulates until the end of the run (actually to the end of each section of a run). This may mean that accuracy is lost, particularly towards the end of long runs, if small increments are added to an already large sum.
`name`	character	The name of an output variable (one word). This is the internal name as used in the model code. A list of available variables is provided in Section 9. This list was correct at the

		time of writing, but the most reliable way to determine exactly which variables are available for a particular version of JULES is to look at the variables listed in the subroutine `init_out_varlist`, and which can be echoed to screen at the start of a JULES run by setting `echo=TRUE` in `INIT_OPTS` (see Section 6.2). A variable may appear more than once in an output profile, as long as each time it appears with a different time flag – e.g. instantaneous and time-average values.
`useName`	character	The name to be used in the output (one word). This variable need not be specified. If `useName` is not provided, the code will substitute `name` instead. This facility allows the user to choose to call output variables by names other than those used in the code, for example to use names that are more memorable, or shorter names to avoid typing! Although the name should be a single word, characters such as underscore ("_") may be used.

6.17.3. Compression of the output grid

As noted above, `outCompress=TRUE` can be used to compress the output data so that any "missing" points are not written and file size is reduced. Although this facility was designed to work with the pdef option of GrADS, it might be useful with other packages too, with the proviso that the user may have to tell another package how to use the available information.

This facility will be described by considering an example in which we have global input data on a 1° grid, and JULES is run at land points only. We would like to visualise the output plotted on the full globe. The input grid is of size 360×180=64800 points, of which only about 25% are land points at which the model is run. If we set `outCompress=TRUE`, the output files will contain data only for the land points, and a mapping is defined so that the land points can be plotted in their correct positions on the Earth. This leads to considerable saving on disc space. The data in the output file is written as a vector (of ~15000 points in this case), in the order that they are held in the model grid. The mapping is written to a binary file that contains 3 fields on the full, expanded grid (360x180 points in this example, starting from the southwest corner, proceeding across each row, then onto next row – i.e. the default JULES order). The first field is integer, and gives the location in the output vector (of ~15000 points) that should be plotted at this location in the globe. If there are no data for a point (i.e. a sea point in this case), the missing data value is inserted. The second field is real, and is 1.0 at points with data, elsewhere 0.0. The third field is not used by JULES (it deals with wind rotation) and will consist of the missing data value.

GrADS' pdef option can be used to display just such a thinned grid, i.e. the "full" grid is populated with values from the "thinned" grid, with missing data values inserted at all other points. Note that `outCompress` as implemented in JULES is a subset of the full pdef available in GrADS, namely where pdef is used with a supplementary file, and each point in the "thinned" output grid maps onto a single point in the "full" grid – effectively there is no interpolation. Thus the latitudes and longitudes of the model gridpoints (specified in `INIT_GRID` above) must be consistent with those specified here for the "full" grid.

If a package other than GrADS is being used to display the thinned data, the user will have to either work out how to use the GrADS mapping between the vector and the full grid, or create new mapping data.

6.17.4. An example of output grids and mapping

This example uses the grids shown in Figure 6. The model grid has `nx`=5, `ny`=4 as shown, and is regular in latitude and longitude. For simplicity, we will assume that the input grid was identical to the model grid. The user wishes to output the 3 shaded points to an output grid with `nxOut`=2, `nyOut`=2, maintaining their relative positions (as given by latitude and longitude).

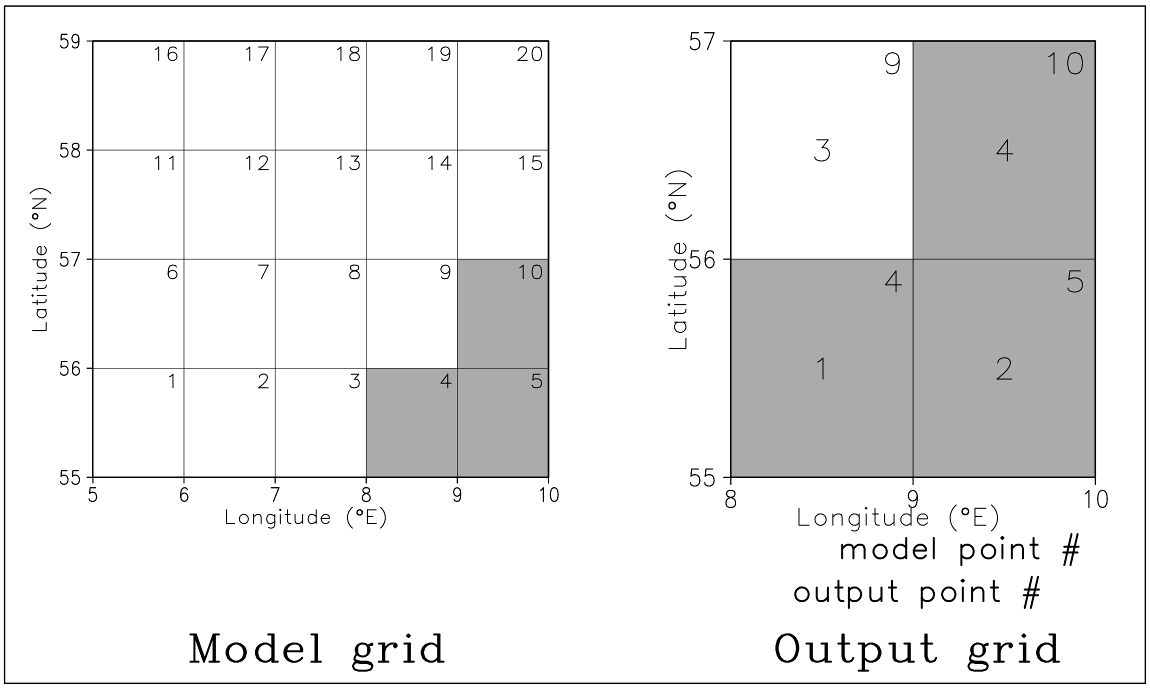

Figure 6. An example of the grids used in output mapping.

The easiest way to achieve this is to use the following lines in the run control file (irrelevant lines have been omitted):

```
2,3             !  pointsFlag(1:2)
F,T             !  outCompress,outLLorder
F               !  readFile

3               !  pointsOut
4,5,10          !  mapOut(1:pointsOut,1)
```

`pointsFlag(1)=1` means that the chosen points will be listed individually.
`pointsFlag(2)=3` means the output grid is to be the smallest rectangle that includes all output points.

`outCompress=F` means the output grid will be padded as necessary. In this case, it means that output point #3 in Figure 6 will be filled with the missing data value.
`outLLorder=T` means the location of each point in the output grid is calculated using the latitude and longitude of the point.
`pointsOut=3` indicates that 3 points are to be output.
`mapOut(1:pointsOut,1)` indicates that the points to be output are numbers 4, 5 and 10 in the input grid (which is identical to the model grid in this case).
The model uses the latitude and longitude of each point to establish that the chosen points should occupy locations 1, 2 and 4 in the output grid, and that location 3 should be filled with the missing data flag (`undefOut`).

The same effect could be achieved by using `pointsFlag(1)=2`, `pointsFlag(2)=2`, `mapOut(:,2)=1, 2, 4`, `outGridNxy=2, 2`, i.e. the user can completely specify the mapping and grid shape. Calculating `mapOut(:,2)` is trivial in this example, but would involve the user in more and unnecessary work if many more points were to be output.

6.17.5. Notes on output

1. A warning is raised if any output is not generated because the output period is never completed. This can occur when a run starts or end partway through an output period, or if a spin up cycle ends partway through an output period. For example, if monthly average diagnostics are requested, but the run ends on the 10th day of a month, the final monthly average is incomplete. In such cases, a value is still written to the output file, but the details of this value vary between cases. In short, a monthly or annual average is calculated if a "large fraction" of the month or year has been simulated, but averages over shorter periods are not calculated and a "missing data" value is output. For details, see the code.
2. GrADS output: A control file (`.ctl` file), that describes GrADS output, includes a specification of the number of times of output that are contained in the associated data files (the TDEF line). When a data file is first opened, a control file is written, with an estimate of the expected number of times that will be written. Sometimes this initial estimate will prove wrong (for example, if the model is spinning up the number of spin up cycles may not be known in advance), and the `.ctl` file is later rewritten when the data file is complete. Under most circumstances, this procedure is carried out without any problem. However, if the user opens the `.ctl` file in GrADS while the integration is still underway, it may not correctly specify the number of times. In that case, the `.ctl` file will be correct if reopened at a later time. However, if the user has moved the `.ctl` file while the integration is underway, it cannot be rewritten and a warning is raised if an attempt is made to rewrite it.
3. Driving data, such as meteorological or vegetation data, may not be correctly represented in output at the start of the first timestep of the run (i.e. time=0), depending upon the frequency of data and any temporal interpolation. The problem arises because the initial output is generated before the procedures that update the driving data are called. Under some circumstances, the driving data will already have been updated during the initialisation, and so the output will be correct. In other cases, the initial output will have "nonsense" values such as zero for the driving data.
4. The code that generates output contains many options and has to deal with a variety of possibilities in terms of output frequency, run dates, spin up and the likes. Until the code has been thoroughly tested by the user community, early versions of JULES are quite likely to contain bugs, particularly in the output code. If a user finds an error with the output, the

bug should be reported, but in the meanwhile JULES will hopefully run correctly if "simpler output" is requested. Two simplifying options, that may not always be practicable for the user, are to request snapshot diagnostics (rather than time averages; in cases of extreme difficulty these snapshots should be every timestep), and to send all output to a single file.

6.18. File name templating

If the names of input files follow particular patterns, JULES can use a substitution template rather than requiring a potentially long list of file names[7]. Templating comes in two forms, time templating and variable name templating, which can be used separately or together.

Valid substitution strings are listed in Table 30. These are 3-character strings, starting with "%". Note that any file name that contains "%" is assumed to use templating.

Table 30. Valid substitution strings for substitution templates.

Substitution string	Description
Time templating	
`%tc`	1-character representation of decade (Met Office files)
`%y4`	4-digit year
`%y2`	2-digit year
`%yc`	1-character representation of year (Met Office files)
`%m2`	2-digit month
`%m1`	1- or 2-digit month
`%mc`	3-character month abbreviation
`%mm`	1-character representation of month (Met Office files)
`%d2`	2-digit day of month
`%d1`	1- or 2-digit day of month
`%dm`	1-character representation of day of month (Met Office files)
`%h2`	2-digit hour of day
`%h1`	1- or 2-digit hour of day
`%hc`	1-character representation of hour of day (Met Office files)
`%n2`	2 digit minute (leading zero if needed)
Variable name templating	
`%vv`	A character variable

6.18.1. Time templating

Information about the time of each file is contained in the file name. Valid substitution strings are listed in Table 30 and examples of the use of time templating are given in Table 32.

[7] JULES templating is similar to that used by GrADS, with a few important differences. JULES only allows a subset of the GrADS substitution strings (not including the %ch string used with chsub), but is more flexible in how it deals with time-templating.

The substitution template must be compatible with the period (frequency) of the data files. If a substitution template includes a substitution string that refers to a period of a day or longer, each file must contain data for no more than one period. For example, if `%m2` appears in the template, each file must contain data from at most one calendar month. For periods less than one day (i.e. hours and minutes), data for more than one period can be held in the same file, but the file period must be a factor of one day[8].

The start time of each file must also follow (slightly complicated) rules that are laid out in Table 31. The rules ensure that the first data in a file represent the first time that the time-templating expects to find in that file. Essentially they require that each file holds all possible data for the time period – there cannot be any missing times. Some of these rules are demonstrated in the example section below. If these rules are not followed, the code will detect an error and stop. In Table 31, `dataPerUnits` and `filePerUnits` are the time units that are used to describe the period of the data and the files respectively, chosen from 1 year, 1 month, days, hours and minutes. If a file or data period can be described by more than one time unit, the longer unit is used. For example, a period of 60 minutes is described as 1 hour.
For example, consider daily data held in one file per month. This gives `dataPerUnits`='day' and `filePerUnits`='1 month'. Table 31 shows that the first data in each file must represent the 1st of the month, as might be expected. A file that started with data for the 2nd of the month cannot be used with time templating, even if a particular run does not require the data at that time.

Table 31. Requirements for the time of first data in time templated files.

		dataPerUnits				
		1 year	1 month	days	hours	minutes
filePerUnits	1 year	none	Jan	01Jan	00H 01Jan	00H 01Jan
	1 month	-	none	1st of month	00H 1st of month	00H 1st of month
	days	-	-	none	00H	00H
	hours	-	-	-	none	00H
	minutes	-	-	-	-	none

6.18.2. Variable-name templating

Variable-name templating is so called because it is expected to be used when related variables are stored in separate files, with file names that are identical apart from a section that indicates what variable is in each file. For example, variable #1 could be in "`file1.dat`", while variable #2 is in "`file2.dat`". Examples of the use of this type of templating are given in the next section. If using variable name templating with non-SDF formats, the layout of each file must be similar – the number of headers and the number of fields in any time level must be the same in all files.

Table 32. Examples of the use of file name templating.

[8] Users of GrADS should note that, for these shorter substitution string periods (hours and minutes), JULES can use files that cannot be described by a GrADS template control file. GrADS (at v1.9v4) insists that each file contains data that covers at most one period, whereas JULES allows data for more than one period. For example, if the substitution template includes %h2, GrADS insists that each file contains data for at most one hour, whereas JULES allows each file to have 1, 2, 3, 4..etc hours of data.

Substitution template	Description of files	Valid template?	Example file names	Comments
/data/met_data_%y4%mc.dat	Monthly files	Yes	/data/met_data_1990jan.dat /data/met_data_1990feb.dat	
./%y4/met_data_%y4%mc.dat	Monthly files	Yes	./1990/met_data_1990jan.dat	A substitution string can appear more than once. Here data for each year are stored in a separate directory.
%vv_%y4	Yearly files, with each variable in a separate file	Yes	Rain_1990.dat Wind_1990.dat	Variable name and time templating used together. The strings that are to be substituted for %vv will be provided by the user via the run control file.
Data_%d2.dat	Hourly data, each file containing data for 10 days	No		Each file can contain at most 1 day of data. For substitution strings that refer to years, months or days, more than one year/month/day of data can be stored in each file.
Data_%h2.dat	Hourly data, each file containing data for 6 hours.	Yes	Data_00.dat Data_06.dat Data_12.dat Data_18.dat	For substitution strings that refer to hours or minutes, more than one hour or minute of data can be stored in each file.
Data_%mc.dat	Hourly data in monthly files. The time of the first data is 00H 01Jun1990.	Yes	Data_jan.dat Data_feb.dat	
Data_%mc.dat	Hourly data in monthly files. The time of the first data is 01H 01Jun1990.	No	Data_jan.dat Data_feb.dat	Similar to the previous case, but with first data one hour later. In this case, the first data in each file must represent 00H on the 1st of a month. These data cannot be described by a time template and instead the name and time of each data file must be listed (see appropriate section).
Data_%y4.dat	Monthly data in yearly files. The time of the first data is given as 00H 15Jan1990.	Yes	Data_1990.dat Data_1991.dat	In this case, the time of the first data must be in January. Here it is shown to be a value at approximately mid-month.

6.19. Notes on temporal interpolation

Time-varying input data to JULES require the user to specify how the data should be interpolated onto the model timestep. The permitted interpolation flags are shown in Table 33. These flags are case-sensitive.

Table 33. Time interpolation flags.

Flag value	Notes
`b`	Backward time average, ending at given time. Will be interpolated with time.
`c`	Centred time average, centred on given time. Will be interpolated with time.
`f`	Forward time average, starting at given time. Will be interpolated with time.
`i`	Instantaneous value at the given time. Will be linearly interpolated with time.
`nb`	Backward time average, ending at given time. Value will be held constant with time.
`nc`	Centred time average, centred on given time. Value will be held constant with time.
`nf`	Forward time average, starting at given time. Value will be held constant with time.

Depending upon the time interpolation flags, driving data may need to be supplied for one or two times that fall before or after the times for the integration. The interpolation scheme implemented in JULES for flags `'b'`, `'c'` and `'f'` is a simplified version of the Sheng and Zwiers (1998)[9] method that conserves the period means of the driving data file. In order to ensure conservation of the average, these flags can be used only if the data period is an even multiple of the model timestep (i.e., if `driveDataPer=2*n*timestep`; `n=1, 2, 3`, ...). In these cases the curve-fitting process tends to produce occasional values near turning points that fall outside the range of the input values. Note that for centred data (flags ‘`c`’ and ‘`nc`’) the time of the data should be given as that at the start of the averaging period, rather than the centre. e.g. the 3-hour average over 06H to 09H, centred at 07:30H, should be treated as having timestamp 06H.

[9] Sheng and Zwiers (1998) “An improved scheme for time-dependent boundary conditions in atmospheric general circulation models”, *Climate Dynamics*, **14**, 609—613.

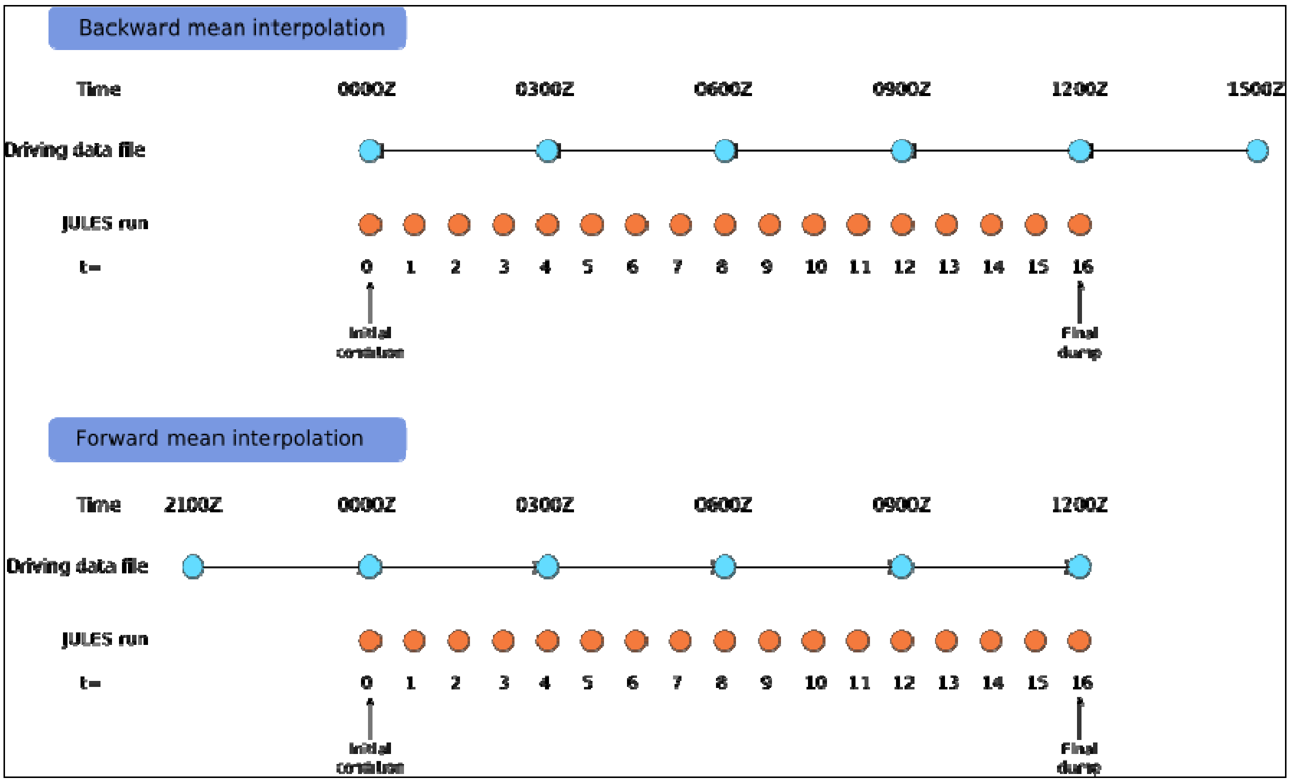

Figure 7. Schematic of JULES interpolation of driving variable from a 3 hour timestep to a 45 minute timestep. Simulation start time is 0000Z (on an arbitrary day) and end time is 1200Z. Blue circles indicate driving data required to complete a JULES simulation from `t=0` to `t=16`. See text for discussion of requirements for driving variables that are forward or backward means.

6.20. Example run control files

Two example run control files come bundled with the JULES source code, in the top-level directory.

`point_loobos_example.jin` for a single point simulation forced with weather station data. This run requires a single input file (meteorological data) that is also included as part of the JULES distribution, in the "LOOBOS" directory. The results of running this code are also provided in the same directory, so the user can check that their installation of JULES produces results that are acceptably close to those of this standard run.

`grid_gswp2_example.jin` for a gridded domain simulation forced with GSWP2 weather data. This run requires a large amount of input data that is not distributed with JULES, and merely serves as an example of a run control file for a gridded domain.

7. Aspects of the code

7.1. Low-level i/o code

In the course of adding to JULES, a user may well want to read new variables into the model. Most of the input/output of spatial fields is handled by subroutines provided by the module READWRITE_MOD. Particularly important procedures that deal with input are summarised in Table 34. To use this code to read in a new variable, the appropriate procedure should be identified based on the type of variable that is to be read in. For example, to read a field that is only defined on land points, a call to readVar2dComp is appropriate. All these procedures require arguments that define the mapping between the input grid and the model grid.

Note that the choice of procedure is governed solely by the type of variable and is not affected by the shape of the input grid. The correct use of these procedures and the arguments required can be learned by studying the exiting code.

Table 34. Key procedures for reading input data.

Name	Summary
`readVar2d`	Reads a variable that is defined at all possible points (both land and sea). The result is a variable on the model grid (this is considered to be a 2-dimensional variable on (x,y), even if the model grid is effectively a vector with ny=1). For example, air temperature is defined at all possible points, both land and sea.
`readVar2dComp`	Reads a variable that is only defined on a subset of points (for example land points). The result is a vector. For example, a land variable can be read from a 2-D (x,y) map (that may contain both land and sea points), and the result is a vector on land points. (The "Comp" in the name is meant to suggest "compression" to a vector!)
`readVar3dComp`	As `readVar2dComp`, but the variable is also a function of the vertical level (e.g. a soil variable on several levels). This 3d version works by looping over the vertical levels, calling the 2d version for each level.

7.2. How to implement new diagnostics for output

The steps needed to add a new diagnostic vary according to what variables are needed in order to calculate the diagnostic. These are covered in the next sections.

7.2.1. Output of existing variables

The data are already held in an existing FORTRAN variable, in a module. This is the easiest case, since the data are easily accessed. The name that is used to select the diagnostic should be added to subroutine `init_out_varlist`, following the existing examples. Care should be taken to specify the correct type of diagnostic (e.g., land points only, soil layers). If the desired diagnostic does not fit any of the existing types, the user may have to closely study the code to work out how to add a new type, and/or contact the JULES developers. Finally, code to load the values into the output space has to be added to subroutine `loadout` (in module `OUTPUT_MOD`). This code may have to calculate the diagnostic using other variables.

7.2.2. Output of new variables

Diagnostics that require variables that the user had added, or that must be calculated in a section of the model code other than the output routines, are more complex to add to JULES. In such a case, it may be easiest to declare a new variable in a FORTRAN module, and to use this variable to hold the values of the diagnostic. Space for the new variable will likely have to be allocated, and the tidiest way to do this would be in the subroutine `allocate_arrays` (which is called at various points during initialisation). The variable can then be accessed by the output procedures and the steps outlined in case 1 above should be followed.

A more sophisticated scheme which only allocated space for a diagnostic if it was required, and loaded the value from any subroutine (avoiding the need to hold the variable in a module, or pass it through the code) has been implemented in some versions of JULES but is not available in the release versions because it was not compatible for use in the Unified Model. If you are keen to get this code, contact the JULES developers.

8. Known limitations of and bugs in the code

1. Spin up over short periods
The current code cannot cope with a spin up cycle that is short in comparison to the period of any input data. For example, a spin up cycle of 1 day cannot use 10-day vegetation data. The code will likely run but the evolution of the vegetation data will probably not be what the user intended! However, it is unlikely that a user would want to try such a run.

2. Lack of more generic i/o code
If a user wants to introduce new time-varying data that cannot be made to fit into the existing code for vegetation or meteorological data (for example, the new data would need to have the same frequency as the other data type), they may have a substantial job on their hands! For many purposes, a simple ‘hack’ may suffice (e.g. write code to read a particular data set for a particular run), but this will lack generality and options such as automatic spin up will be hard to accommodate. At present there is no good solution – we don’t have any flexible coupling code that can be told to fetch suitable values of an arbitrary field, although JULES may move towards this in future.

9. Variables available for output

Variables that are available for output from JULES are listed in the tables of this section, separated according to their type. Types of variables are:

SINGLE: a single value at all gridpoints (land and sea) (Table 35).
LAND: a single value at land gridpoints (Table 36).
PFT: a value for each of `npft` PFTs at each land gridpoint (Table 37).
TILE: a value for each of `ntiles` tiles at each land gridpoint (Table 38).
TYPE: a value for each of `ntype` surface types at each land gridpoint (Table 39).
SOIL: a value for each of `sm_levels` soil layers at each land gridpoint (Table 40).

These tables were correct at the time of writing, but the most reliable way to determine exactly which variables are available for a particular version of JULES is to look at the variables listed in the subroutine `init_out_varlist`, and which can be echoed to screen at the start of a JULES run by setting `echo=TRUE` (see Section 6.2).

Table 35. A list of output variables that have a single value at each gridpoint.

Name	Description
conRain	Gridbox convective rainfall (kg m^{-2} s^{-1})
conSnow	Gridbox convective snowfall(kg m^{-2} s^{-1})
cosz	Cosine of the zenith angle (-)
diffFrac	Gridbox fraction of radiation that is diffuse (-)
ecan	Gridbox mean evaporation from canopy/surface store (kg m^{-2} s^{-1})
ei	Gridbox sublimation from lying snow or sea-ice (kg m^{-2} s^{-1})
esoil	Gridbox surface evapotranspiration from soil moisture store (kg m^{-2} s^{-1})
fqw	Gridbox moisture flux from surface (kg m^{-2} s^{-1})
ftl	Gridbox surface sensible heat flux (W m^{-2})
landAlbedo1	Gridbox albedo for waveband 1 (direct beam visible)
landAlbedo2	Gridbox albedo for waveband 2 (diffuse visible)
landAlbedo3	Gridbox albedo for waveband 3 (direct beam NIR)
landAlbedo4	Gridbox albedo for waveband 4 (diffuse NIR)
latentHeat	Gridbox surface latent heat flux (W m^{-2})
latitude	Gridbox latitude (^{o})
longitude	Gridbox longitude (^{o})
lsRain	Gridbox large-scale rainfall (kg m^{-2} s^{-1})
lsSnow	Gridbox large-scale snowfall (kg m^{-2} s^{-1})
lwDown	Gridbox surface downward LW radiation (W m^{-2})
precip	Gridbox precipitation rate (kg m^{-2} s^{-1})
pstar	Gridbox surface pressure (Pa)
q1p5m	Gridbox specific humidity at 1.5m height (kg kg^{-1})
qw1	Gridbox specific humidity (total water content) (kg kg^{-1})
rain	Gridbox rainfall rate (kg m^{-2} s^{-1})
snomltSurfHtf	Gridbox heat flux used for surface melting of snow (W m^{-2})
snowfall	Gridbox snowfall rate (kg m^{-2} s^{-1})
snowMass	Gridbox snowmass (kg m^{-2})
surfHtFlux	Gridbox net downward heat flux at surface over land and sea-ice fraction of gridbox (W m^{-2})
swDown	Gridbox surface downward SW radiation (W m^{-2})
t1p5m	Gridbox temperature at 1.5m height (K)
taux1	Gridbox westerly component of surface wind stress (N m^{-2})
tauy1	Gridbox southerly component of surface wind stress (N m^{-2})
tl1	Gridbox ice/liquid water temperature (K)
tstar	Gridbox surface temperature (K)
u1	Gridbox westerly wind component (m s^{-1})
u10m	Gridbox westerly wind component at 10 m height (m s^{-1})
v1	Gridbox southerly wind component (m s^{-1})
v10m	Gridbox southerly wind component at 10m height (m s^{-1})

Table 36. A list of output variables that have a single value at each land gridpoint.

Name	Description
`canopy`	Gridbox canopy water content (kg m^{-2})
`cs`	Gridbox soil carbon (kg C m^{-2})
`cv`	Gridbox mean vegetation carbon (kg C m^{-2})
`depthFrozen`	Gridbox depth of frozen ground at surface (m)
`depthUnfrozen`	Gridbox depth of unfrozen ground at surface (m)
`gpp`	Gridbox gross primary productivity (kg C m^{-2}s^{-1})
`gs`	Gridbox surface conductance to evaporation (m s^{-1})
`hfSnowMelt`	Gridbox snowmelt heat flux (W m^{-2})
`landIndex`	Index (gridbox number) of land points
`liceIndex`	Index (gridbox number) of land ice points
`litCMn`	Gridbox mean carbon litter (kg C m^{-2} (360days)$^{-1}$)
`lwUp`	Gridbox surface upward LW radiation of land points (W m^{-2}) (assuming an emissivity of 1)
`npp`	Gridbox net primary productivity (kg C m^{-2} s^{-1})
`radnet`	Surface net radiation of land points (W m^{-2})
`respP`	Gridbox plant respiration (kg C m^{-2} s^{-1})
`respS`	Gridbox soil respiration (kg C m^{-2} s^{-1})
`respSDrOut`	Gridbox mean soil respiration for driving TRIFFID (kg C m^{-2} (360days)$^{-1}$)
`runoff`	Gridbox runoff rate (kg m^{-2} s^{-1})
`smcAvailTop`	Gridbox available moisture in top 0.0m of soil (kg m^{-2})
`smcAvailTot`	Gridbox available moisture in soil column (kg m^{-2})
`smcTot`	Gridbox total soil moisture in column (kg m^{-2})
`snowCan`	Gridbox snow on canopy (kg m^{-2})
`snomltSubHtf`	Gridbox sub-canopy snowmelt heat flux (W m^{-2})
`snowFrac`	Gridbox snow-covered fraction of land points (-)
`snowGround`	Gridbox snow below canopy (kg m^{-2})
`snowMelt`	Gridbox rate of snowmelt (kg m^{-2} s^{-1})
`soilIndex`	Index (gridbox number) of soil points
`subSurfRoff`	Gridbox sub-surface runoff (kg m^{-2} s^{-1})
`surfRoff`	Gridbox surface runoff (kg m^{-2} s^{-1})
`surfRoffInf`	Gridbox infiltration excess surface runoff (kg m^{-2} s^{-1})
`swetLiqTot`	Gridbox unfrozen soil moisture as fraction of saturation (-)
`swetTot`	Gridbox soil moisture as fraction of saturation (-)
`Tfall`	Gridbox throughfall (kg m^{-2} s^{-1})
`Trad`	Gridbox effective radiative temperature (K) (assuming an emissivity of 1)
`wFluxSfc`	Gridbox downwards moisture flux at soil surface (kg m^{-2} s^{-1})

Table 37. A list of output variables that have a single value for each PFT at each land gridpoint.

Name	Description
`cVegP`	PFT total carbon content of the vegetation (kg C m^{-2})
`canhtP`	PFT canopy height (m)
`ciP`	PFT internal CO2 pressure (Pa)
`fsmcP`	PFT soil moisture availability factor (-)
`gLeafP`	PFT leaf turnover rate ($[360days]^{-1}$)
`gLeafDayP`	PFT mean leaf turnover rate for input to PHENOL ($[360days]^{-1}$)
`gLeafDrOutP`	PFT mean leaf turnover rate for driving TRIFFID ($[360days]^{-1}$)
`gLeafPhenP`	PFT mean leaf turnover rate over phenology period($[360days]^{-1}$)
`gstomP`	PFT bulk (canopy) stomatal conductance for water vapour (m s^{-1})
`gppP`	PFT gross primary productivity (kg C m^{-2} s^{-1})
`laiP`	PFT leaf area index (-)
`laiPhenP`	PFT leaf area index after phenology (-)
`litCP`	PFT carbon litter (kg C m^{-2} $(360days)^{-1}$)
`nppDrOutP`	PFT mean NPP for driving TRIFFID (kg C m^{-2} $(360days)^{-1}$)
`nppP`	PFT net primary productivity (kg C m^{-2} s^{-1})
`rdcP`	Canopy dark respiration, without soil water dependence (mol CO_2 m^2 s^{-1})
`respPP`	PFT plant respiration (kg C m^{-2} s^{-1})
`respWDrOutP`	PFT mean wood respiration for driving TRIFFID (kg C m^{-2} $(360days)^{-1}$)

Table 38. A list of output variables that have a single value for each tile at each land gridpoint.

Name	Description
alb1T	Tile land albedo, waveband 1 (direct beam visible)
alb2T	Tile land albedo, waveband 2 (diffuse visible)
alb3T	Tile land albedo, waveband 3 (direct beam NIR)
alb4T	Tile land albedo, waveband 4 (diffuse visible)
canopyT	Tile surface/canopy water for snow-free land tiles (kg m^{-2})
catchT	Tile surface/canopy water capacity of snow-free land tiles (kg m^{-2})
ecanT	Tile evaporation from canopy/surface store for snow-free land tiles (kg m^{-2} s^{-1})
eiT	Tile sublimation from lying snow for land tiles (kg m^{-2} s^{-1})
esoilT	Tile surface evapotranspiration from soil moisture store for snow-free land tile (kg m^{-2} s^{-1})
fqwT	Tile surface moisture flux for land tiles (kg m^{-2} s^{-1})
ftlT	Tile surface sensible heat flux for land tiles (W m^{-2})
gcT	Tile surface conductance to evaporation for land tiles(m s^{-1})
leT	Tile surface latent heat flux for land tiles (W m^{-2})
q1p5mT	Tile specific humidity at 1.5m over land tiles (kg kg^{-1})
radnetT	Tile surface net radiation (W m^{-2})
rgrainT	Tile snow grain size (μm)
snowCanT	Tile lying snow (on canopy) (kg m^{-2})
snowCanMeltT	Tile melt of snow (on canopy) (kg m^{-2} s^{-1})
snowGroundMeltT	Tile melt of snow under canopy (kg m^{-2} s^{-1})
snowGroundT	Tile snow below canopy (kg m^{-2})
snowMassT	Tile lying snow (total) (kg m^{-2})
snowMeltT	Tile total snow melt (kg m^{-2} s^{-1})
t1p5mT	Tile temperature at 1.5m over land tiles (K)
tstarT	Tile surface temperature (K)
Z0T	Tile surface roughness (m)

Table 39. A list of output variables that have a single value for each tile type at each land gridpoint.

Name	Description
frac	Fractional cover of each surface type.
tileIndex	Index (gridbox number) of land points with each surface type

Table 40. A list of output variables that have a single value for each soil level at each land gridpoint.

Name	Description
`bSoil`	Clapp-Hornberger exponent for each soil layer (-)
`ext`	Extraction of water from each soil layer (kg m^{-2} s^{-1})
`hCapSoil`	Soil heat capacity (J K^{-1} m^{-3}) for each soil layer
`hConSoil`	Soil thermal conductivity (W m^{-1} K^{-1}) for each soil layer
`satCon`	Saturated hydraulic conductivity (kg m^{-2} s^{-1}) for each soil layer
`sathh`	Saturated soil water pressure (m) for each soil layer
`smcl`	Moisture content of each soil layer (kg m^{-2})
`soilWet`	Total moisture content of each soil layer, as fraction of saturation (-)
`sthf`	Frozen moisture content of each soil layer as a fraction of saturation (-)
`sthu`	Unfrozen moisture content of each soil layer as a fraction of saturation (-)
`tSoil`	Sub-surface temperature of each layer (K)
`vsmcCrit`	Volumetric moisture content at critical point for each soil layer (-)
`vsmcSat`	Volumetric moisture content at saturation for each soil layer (-)
`vsmcWilt`	Volumetric moisture content at wilting point for each soil layer (-)
`wFlux`	Downwards moisture flux at bottom of each soil layer (kg m^{-2} s^{-1})

10. List of Tables