

LPJ-GUESS Reference – 2013-07-11

Ben Smith, Joe Siltberg

Revision history for this document

Guy Schurgers	2011-03-02	Added BVOC
Mats Lindeskog	2011-01-19	Added the new Gridcell class
Michael Mischurow	2012-07-03	Diurnal processes in the canopy exchange module
David Wårlind	2013-03-20	Added N cycle with new SOM dynamics
Joe Siltberg	2013-07-11	Input/Output parts rewritten due to new structure of the code
Joe Siltberg	2014-06-26	Various minor technical updates (how to compile etc.)

BUILDING AND RUNNING

The easiest way to build LPJ-GUESS is to use CMake. CMake is not a compiler itself, instead it generates a native build system for your platform of choice. LPJ-GUESS has been built and tested with CMake on the following platforms:

- Microsoft Windows with Visual Studio 2008
- Microsoft Windows with Visual Studio 2013
- Linux with the GNU C++ compiler
- Linux with the Intel C++ compiler

In theory it should work on other platforms supported by CMake such as Borland or MinGW.

Installing CMake

CMake can be downloaded for free from <http://www.cmake.org/> . There are installation packages for Windows, Mac OSX, Linux and other Unix operating systems. Download a package for your system and follow the installation instructions.

To find out if CMake is already installed on your system, run the following command:

```
cmake --version
```

If you don't have access rights to install software on your system, CMake can be run from a directory you do have access to.

During installation of CMake you will be asked whether you wish to add CMake to the system PATH. It is recommended that you do so.

Compiling on Windows with Visual Studio

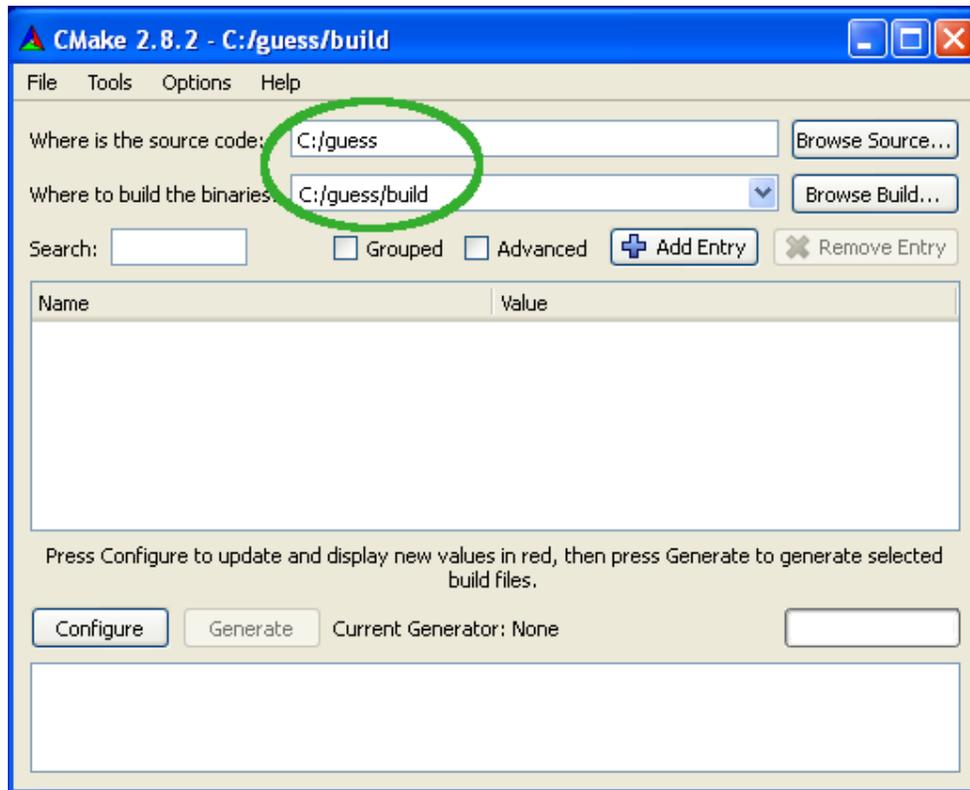
The following instructions apply for Visual Studio 2013, other versions should be similar. On Windows LPJ-GUESS can either be run as a command line program (text based) or in the graphical Windows Shell where plots are displayed as output is generated.

Creating Visual Studio project files

After you've made sure CMake is installed, the first step is to use CMake to create project files for your version of Visual Studio.

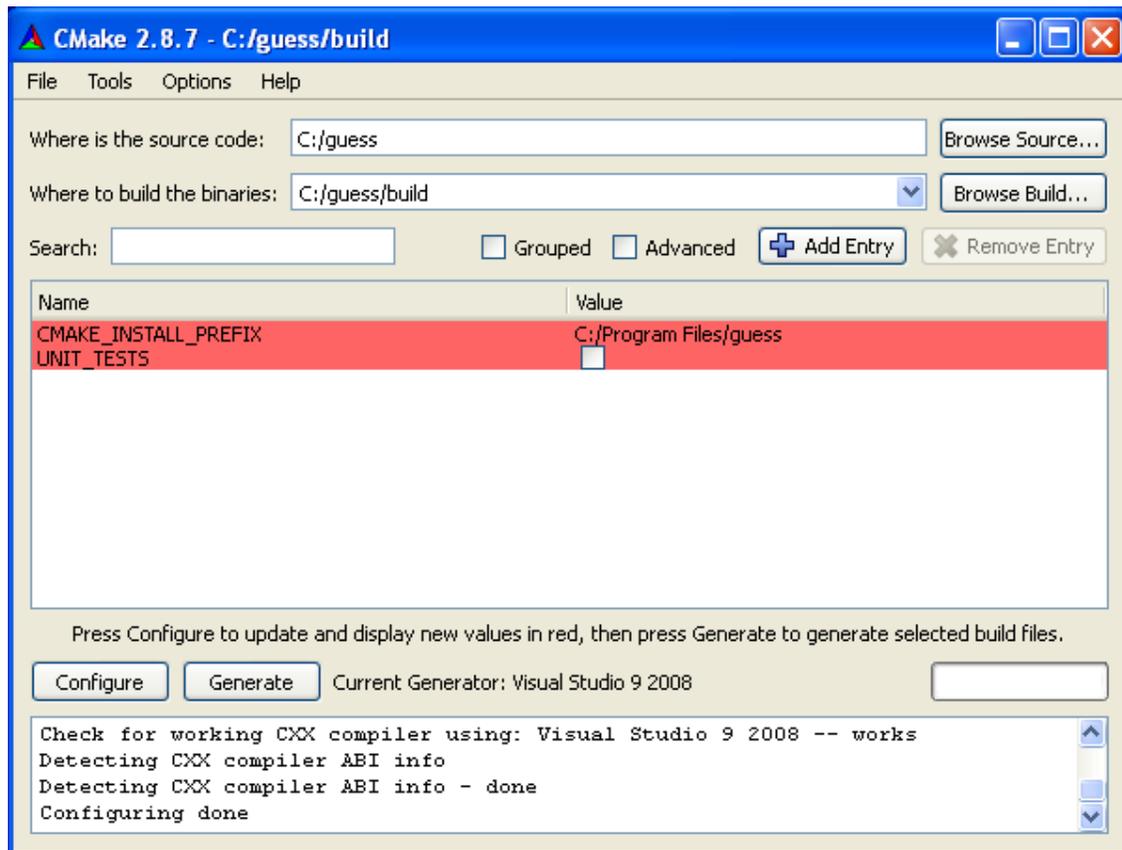
- Start the program cmake-gui (look under "All programs" in your Start menu)
- Enter the path to the directory containing all the LPJ-GUESS files (**NB:** This should be the root of your LPJ-GUESS directory, see example below)
- Enter a path to a directory where you want to place Visual Studio files, for instance the build directory in the LPJ-GUESS directory

It should look something like this:



Click on the **Configure** button to choose your version of Visual Studio. After you've made your choice and clicked Finish, cmake will try to find your installed C++ compiler and some other things. When that's done you get a chance to configure how to compile LPJ-GUESS.

It should look something like this:



At this point you can make changes in how the project should be constructed. The available configuration settings vary depending on your system. Hover with your mouse over one of the configuration options to get a description of what it does. After you're done press **Configure** again to use the chosen settings. You can then press **Generate** to create the Visual Studio project files, they will end up in the build directory you chose earlier.

Compiling

Open LPJ-GUESS in Visual Studio by double clicking on one of the files generated by CMake. If you're using Visual Studio 2013 you should open the solution file, called guess.sln.

In Visual Studio you should see at least two projects, one called guess and one called guesscmd. The first is for compiling the graphical Windows Shell version, the other is for the command line version. There is also a project called ALL_BUILD which forces compilation of both guess and guesscmd.

Hit F7 to build everything (or choose Build from the Build menu), there shouldn't be any errors.

Configurations (Debug/Release)

You can choose to build your program in different Visual Studio configurations. The most important ones are called Debug and Release. The Debug configuration is the default and should be used during development since it allows you to find problems in the code with the built-in debugger. The Release configuration results in much faster programs however and should be used when you run your model for real.

How to switch configuration is different in different versions of Visual Studio. In Visual Studio 2013, use the “Configuration Manager...” in the Build menu.

The compiled program will be placed in your build directory under a directory named as the configuration you’re using. So if you have your Visual Studio files under C:\guess\build, and you’re using the Debug configuration, you will find the compiled LPJ-GUESS program under C:\guess\build\Debug.

Running

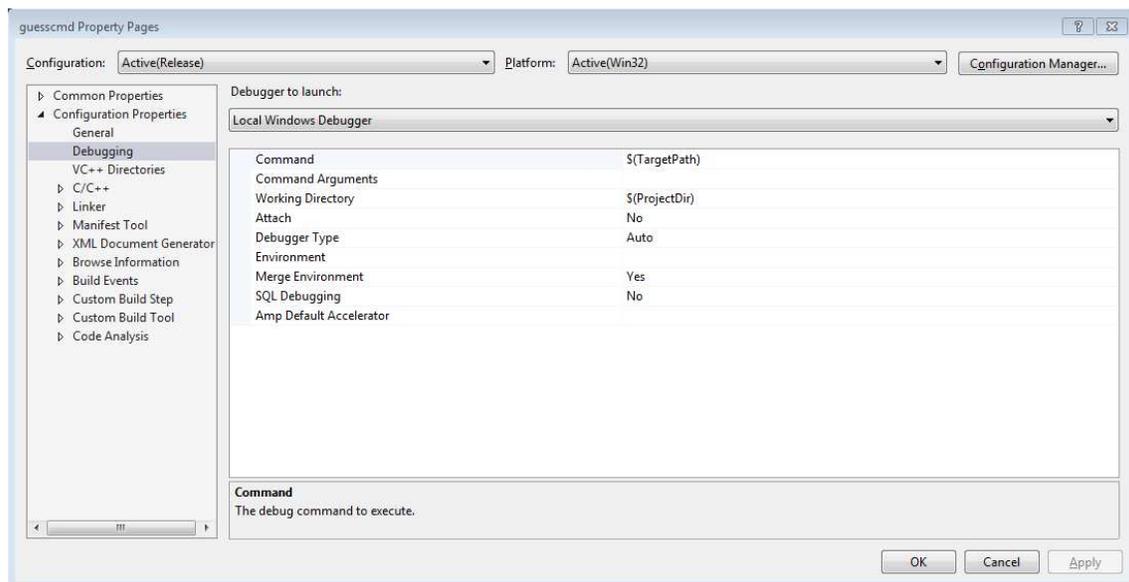
Command line

To run the command line version from within Visual Studio you need to choose a Working Directory and Program Arguments.

The Working Directory is the directory where the program will run, any relative paths that you have in the instruction files are relative to this directory. Some output files may be placed in this directory, for instance the guess log file.

The Program Arguments should contain the path to the instruction file you wish to use (absolute path or relative to the Working Directory.)

How to configure these depends on your Visual Studio version. In Visual Studio 2013, right click on the guesscmd project and choose “Properties”. Find “Debugging” in the tree in the left hand pane:



To start the program, make guesscmd the “StartUp project” (by right clicking on the guesscmd project) and hit F5 (for debugging) or Ctrl+F5 (runs as a regular program).

Windows Shell

You’ll find the Windows Shell program at windows_version\guesswin.exe in your LPJ-GUESS directory. Copy the guesswin.exe file to the directory where Visual Studio has compiled LPJ-GUESS (for instance build\Debug if you’re using the Debug configuration). If the program compiled successfully you should have a file named guess.dll in this directory.

By default, if you ask Visual Studio to run the project, it will try to run the guess.dll file. But since that is only a library used by guesswin.exe, we have to tell Visual Studio to start the exe instead. Do this by again going to the “Debugging” settings (right click on the guess project, choose “Properties” and find the “Debugging” section in the left hand pane). Set the “Command” property to the file path to guesswin.exe you just placed next to guess.dll. You don’t have to specify “Command Arguments” or “Working Directory” when using the Windows Shell.

To start the program, make guess the “StartUp project” (by right clicking on the guesscmd project) and hit F5 (for debugging) or Ctrl+F5 (runs as a regular program).

You can also run the model directly from outside Visual Studio by executing (e.g. by double-clicking) guesswin.exe. Again, it has to be the guesswin.exe in the directory where your guess.dll is.

Adding new source code files

To add new source code files to the project, edit one of the CMakeLists.txt files. Each directory with source code has its own CMakeLists.txt, so if you want to add a file under modules, edit the modules\CMakeLists.txt file. In it you will find a list of the headers and cpp files, simply add your new file(s) to the appropriate list(s).

The next time you build your project the Visual Studio project files should be updated automatically. Once they’ve been updated you can build again to actually compile with your new file(s).

Compiling on Unix systems

Creating build files

Start by using CMake to create build files and configure how you want to compile LPJ-GUESS. You can create the build files anywhere you like, for instance in the build directory in the LPJ-GUESS directory. From that directory, run the command ccmake with the path to the LPJ-GUESS directory as an argument, for instance:

```
ccmake ..
```

or (if you’re not in the build directory):

```
ccmake /path/to/my/lpjguess
```

This will start a configuration tool where you can choose some settings. Press ‘c’ to get a list of the settings you can change. Use the arrow keys to navigate among the settings. Each setting has a description, shown in the lower part of the screen when you select it (you may want to increase the size of your terminal to read it properly.) When you’re done, hit ‘c’ again and then ‘g’ to generate the build files.

If you know that you will use the default settings in the configuration tool, you can skip this step by running `cmake` rather than `ccmake`.

Compiling

After the project has been configured, stay in the directory where the build files were generated and run the following command:

```
make
```

When `make` finishes, you should have an executable program named “guess” in the same directory. You should also get a file named “submit.sh” which is used when running on parallel clusters.

Running

The generated program can be started either from the directory where it was created, or copied to where you want to run your model. Run it by supplying the instruction file as an argument, for instance:

```
/home/sally/lpguess/build/guess myinstructionfile.ins
```

or, start with `-help` to get a list of the available instruction file parameters:

```
/home/sally/lpguess/build/guess -help
```

These commands start `guess` as a regular program. For running on parallel clusters, see below.

Adding new source code files

To add new source code files to the project, edit one of the `CMakeLists.txt` files. Each directory with source code has its own `CMakeLists.txt`, so if you want to add a file under `modules`, edit the `modules\CMakeLists.txt` file. In it you will find a list of the headers and `cpp` files, simply add your new file(s) to the appropriate list(s).

The build files will automatically update the next time you run `make`.

Changing compiler

The easiest way to change compiler is to supply your chosen compiler on the command line when running `ccmake`, for instance, to choose to compile with the Intel C++ compiler (called `icc`):

```
CXX=icc ccmake ..
```

Compiler settings

To change the compiler settings, edit the CMakeLists.txt file in the root of your LPJ-GUESS directory. Look for the CMAKE_CXX_FLAGS variable. For information on available compiler flags, see the reference manual for the compiler used.

Misc CMake

Using NetCDF

On a Unix system with NetCDF installed, CMake will attempt to link LPJ-GUESS with the NetCDF libraries, and add suitable compiler flags automatically. If you don't have the possibility to install NetCDF, or you wish to use a different version, you specify a path to your own NetCDF installation, for instance in your home directory. On Windows you always need to specify the path to your NetCDF installation since there is no standard location.

This is done either by setting the environment variable CMAKE_PREFIX_PATH to your NetCDF installation before running CMake, or by switching to advanced mode in the CMake interface and specifying each required NetCDF related path.

For instance, to set the environment variable just before running CMake (on a Unix system):

```
CMAKE_PREFIX_PATH=/home/sally/netcdf ccmake ..
```

Eclipse and other generators

CMake can generate different kinds of build files and project files. To see the generators available on your platform, run cmake without arguments.

To specify a generator from the command line, make sure to enclose the generator name in quotation marks (since they usually contain spaces). For instance, to use one of the Eclipse generators:

```
ccmake -G "Eclipse CDT4 - Unix Makefiles" /home/sally/LPJ-GUESS
```

If you're using the generator for Eclipse, make sure your build folder is outside of your LPJ-GUESS folder, this is a known limitation in the Eclipse projects generated by CMake.

Organising your files

LPJ-GUESS lets you set up your model runs in many different ways. If you're not sure where to place your instruction files, gridlists etc., this section presents one way of working which should work well for most users.

Let's say you're working on "Project X", to run certain simulations with LPJ-GUESS perhaps with some modifications of the source code. We'll keep the files for this project in a folder, for instance C:\projectx.

Place your code in a sub folder of projectx. For instance C:\projectx\guess. Especially if you're getting the code from the version control system it will be convenient to have the code in a folder separate from other files such as instruction files and gridlists for your model runs.

When configuring your project with cmake, choose to use the build folder C:\projectx\guess\build.

Under C:\projectx you can create one sub-folder for each type of model run you will have. In that folder you'll place the instruction file for that model run, and the output files will be generated there. It should now look something like this:

- projectx
 - guess
 - scandinavia
 - guess.ins
 - sweden
 - guess.ins
 - sweden_deterministic
 - guess.ins

You may also want to place the gridlist for each model run in its folder. In that case you can use a relative path for the gridlist in the instruction file, such as:

```
param "file_gridlist" (str "gridlist.txt")
```

However, you may find that different model runs use the same gridlist, or that you re-use gridlists across different projects. In that case you may find it more convenient to have a separate folder with gridlists, and to specify absolute paths in your instruction files:

```
param "file_gridlist" (str "C:\gridlists\sweden.txt")
```

Similarly you will probably prefer to place forcing data etc. in separate folders and use absolute paths for those files in the instruction file.

Compiling and running on parallel clusters

Running the model on a cluster often means submitting a job to some kind of queue system. There may also be system dependent steps involved such as copying files to or from the local compute nodes where the program is running. The build system can generate a script for you which takes care of this, but that script is generated for a specific system.

The SYSTEM configuration

When you run cmake you can choose to configure the SYSTEM variable. If you leave it empty the build system will generate a submit script for INES' internal cluster Simba.

If you need to run on a system which isn't supported you need to create your own submit script. The script generated for Simba can be a good starting point.

Adapting your job

The following requirements must be met **exactly**. If not, the job will not work as expected or may not work at all.

Requirement 1: gridlist file

Grid cells / site coordinates **must be** read from one plain-text gridlist file (one row per grid cell). Other solutions, e.g. a lat-lon window, will not work with these tools. Your best bet is to modify your setup and implement a gridlist file.

Requirement 2: gridlist must end with newline

Ensure that the last line of the gridlist file ends with a "newline" character, otherwise the last grid cell will not be simulated. If you are unsure whether this is the case, add a blank line at the end.

Requirement 3: there must be an ins file

Requirement 4: output files must be plain text files

All output files must be plain text files with or without one leading row containing a header or column labels (or a blank first row). If your output files do not meet these conditions, they will still appear for subsets of gridcells in the runX subdirectory for each process (see Running your job) but combined files containing output for all gridcells will be corrupted or may not be created.

Requirement 5: file names must be specified the correct way

File names in the ins file and/or input/output module (guessio.cpp or equivalent) must be specified in the correct way: some files **must** be specified as a **simple file name** with **no directory part** while others **must** be specified as a **full path name including a directory part**.

The rules are as follows. **Read carefully and follow exactly.**

- Full absolute pathnames must be given for all **input** files the model requires (NB: except the gridlist file, see next point), whether these are specified in the ins file or in the code:

OK:

```
param "file_cru" (str "/home/ben/archive/env/cru/cru_1901-1998.bin")
file_soil="/home/ben/archive/env/cralee/soils_lpj.dat";
```

NOT OK:

```
param "file_cru" (str "cru_1901-1998.bin")
file_soil="../guess/soils_lpj.dat";
```

- The gridlist file name must be given as a **simple file name** with **no directory part**, whether it is specified in the ins file or in the code. However, a full or relative pathname for the gridlist file may be specified, if required, in the submit script (see Running your job):

OK:

```
param "file_gridlist" (str "gridlist.txt")
file_gridlist="gridlist.txt";
```

NOT OK:

```
param "file_gridlist" (str "/home/ben/guess/gridlist.txt")
file_gridlist="../gridlist.txt";
```

- **Output** file names are specified as **file names only** with **no directory part**, whether these are specified in the ins file or in the code:

OK:

```
param "file_cmass" (str "cmass.out")
file_flux="flux.out";
```

NOT OK:

```
param "file_cmass" (str "/scratch/fred/guessrun/cmass.out")
file_flux="../guessrun/flux.out"
```

Running your job

1. Create or go to a directory in which to run the model. This will be called the **run directory**. Some systems might have recommendations about where to place such a directory. It needs to be a place which is accessible from the compute nodes. On Simba for instance you should place it under your own directory under /scratch, **not** under /home. There is a workspace with your user name under /scratch:

```
cd /scratch/sally
mkdir guessrun
cd guessrun
```

2. When the model was compiled, a file named "submit.sh" should have been generated as well. Copy it to the run directory:

```
cp /home/sally/lpjguess/build/submit.sh .
```

(don't forget the '.' at the end!)

3. Open submit.sh in a text editor and set appropriate values for the following variables:
 - NPROCESS
 - WALLTIME
 - INSDIR
 - INPUT_MODULE
 - GRIDLIST
 - OUTFILES

The meaning of these variables is documented in the submit script.

4. Start the job:

```
./submit.sh
```

Once the job starts it will create a number of subdirectories called runX (X=process number), one for each process/node in the parallel job. A roughly equal chunk of the gridlist file and a copy of the ins file appears in each subdirectory. Output appears in each subdirectory (for the gridcells listed in the gridlist file there) and, at the end of the run, in the run directory (all gridcells).

You can specify extra arguments to the submit script, to control the name of your job, or to override some of the variables above. For more information, see the documentation in the submit script for your system.

Checking progress etc.

A log file for each subprocess (guess.log) appears in its corresponding runX subdirectory. You can display the last few lines of each log file to check progress by entering the following command in the run directory:

```
sh progress.sh
```

The queue system used on the cluster should also let you query the status of your job. On a cluster with PBS (like Simba), you can run one of these commands:

```
qstat
qstat -u <user-name>
qstat -f <job-id>
```

The first shows all queued and running jobs. The second shows your jobs. The last shows more details about one job. When you submit your job the submit script will print out the job-id.

To cancel a running job, delete it using:

```
qdel <job-id>
```

Final output appears in the run directory. A log with some additional messages appears in files called something like guess.oXXX and guess.eXXX in the run directory, where XXX is the job id of your job. In the event of errors these files may provide some clues.

Troubleshooting

If it doesn't work:

- Carefully reread Adapting your job above, especially requirement 5
- Carefully reread Running your job above
- Check for error messages in guess.oXXX and guess.eXXX, see Checking progress etc.

Specific problems and possible causes:

PROBLEM: Doesn't seem particularly fast for a supercomputer. I am trying this out for one grid cell.

POSSIBLE CAUSE: There is no point running a parallel job for one grid cell, since the grid cells are divided equally between processes (nodes).

PROBLEM: Seems very slow, same grid cells appear several times in output.

POSSIBLE CAUSE: Path name specified for gridlist file in ins file or i/o module. File name only (no directory part) must be specified. See requirement 5 above.

PROBLEM: Runs for several hours then stops before all grid cells have been processed.

POSSIBLE CAUSE: Maximum time for job exceeded. You need to specify a longer wall clock time in the submit script. See Running your job.

PROBLEM: Job seems to run and finish normally, but no output files appear in the run directory.

POSSIBLE CAUSE: Incorrect or missing output file list in submit script. Should be a space-delimited list of file names including extensions and with no directory part. See Running your job.

PROBLEM: Output files are unreadable or corrupted.

POSSIBLE CAUSE: The model (input/output module) is producing output files in the wrong format. Output files must be plain text files with or without ONE initial row containing header information, column labels or a blank line. See requirement 4 above.

PROBLEM: Output is missing for the last grid cell in the gridlist file.

POSSIBLE CAUSE: The last line of the gridlist file does not end with a newline character (see requirement 2 above). Add a blank line at the end of the gridlist file.

PROBLEM: Job seems to enter the batch queue but crashes immediately after start.

POSSIBLE CAUSE: Format errors or incorrect file names specified in submit.sh. (NB: no space allowed after "=" in variable assignments!) See Running your job.

PROBLEM: Job appears in batch queue but never seems to start.

POSSIBLE CAUSE: You are requesting more processes than there are nodes available. Your job will start when there is one free node for each process. Simba has (at the time of writing) around 90 nodes, but some may be reserved/in use by other users. Try requesting fewer processes for your job. See Running your job.

LPJ-GUESS ARCHITECTURE

Architecture of LPJ-GUESS

LPJ-GUESS is designed to provide a flexible platform for modelling ecosystem structural dynamics and functioning. The version described here is a combined implementation of two previously-developed dynamic ecosystem models – LPJ-DGVM (Sitch et al 2000, 2003; Bachelet et al. 2003; Cramer et al. 2001, 2004; Gerten et al. 2004) and GUESS (Smith et al 2001, Hickler et al 2004). In principle, other model formalisms could be constructed with only limited changes to the overall architecture, in terms of data structures and flow control.

This is possible for two reasons. Firstly, the model code is organised into well-defined modules. Each module (with the exception of modules having a primarily "technical" function – for example, input and output of driving data and results) encompasses a relatively well-defined, related subset of ecosystem processes with a distinct spatial and/or temporal signature. A central framework, containing all explicit loops through space (stands, patches) and time (days, years) in the model, binds the modules together and manages the exchange of data between them. The modular structure is designed to reflect the ways in which different ecosystem components and functions are actually linked in nature, making it largely independent of the assumptions and generalisations of any particular model. The modules, framework and links between them are described below. The main links between modules and the framework are shown schematically in Figure 2.

Secondly, data (driving environmental data, ecosystem state variables, static parameters for PFTs, soils etc) are exchanged between modules via hierarchically-organised 'objects' (compound data structures, or classes) which correspond to real ecosystem components – stands, patches, soils, individual plants etc. Whereas the specific variables associated with each object, and parameterisations of the processes acting on them, might vary for different models, the objects themselves, and the hierarchical links between them, should not. The classes in which the main objects are defined are described in detail in the section "LPJ-GUESS classes", below.

The framework

The framework (Figure 1) is the "mission control" of the model. It:

- contains *all* explicit loops through space (grid cells/localities, stands, patches) and time (days, years) in the simulation. Timing loops are nested so that there is no "time travel" – *no process is ever fed information about a particular day or year until that day or year is reached in the hierarchy of timing loops*. This is implemented in the framework function in `framework.cpp`.
- defines the main objects (classes such as Pft, Individual, Climate, Soil etc; see section "LPJ-GUESS Classes", below) and global variables (e.g. the number of years to simulate, the number of replicate patches in each stand, patch size, the simulation time step).
- defines the global utility functions `dprintf`, `fail`, `plot` etc, which are available throughout the model code.
- performs most calls to process functions within the modules, at the appropriate stage in the simulation (a few such calls are delegated to the modules themselves), and thereby manages exchange of data between the modules.
- contains additional functionality which may be needed by different modules, for instance generic mathematical functions.

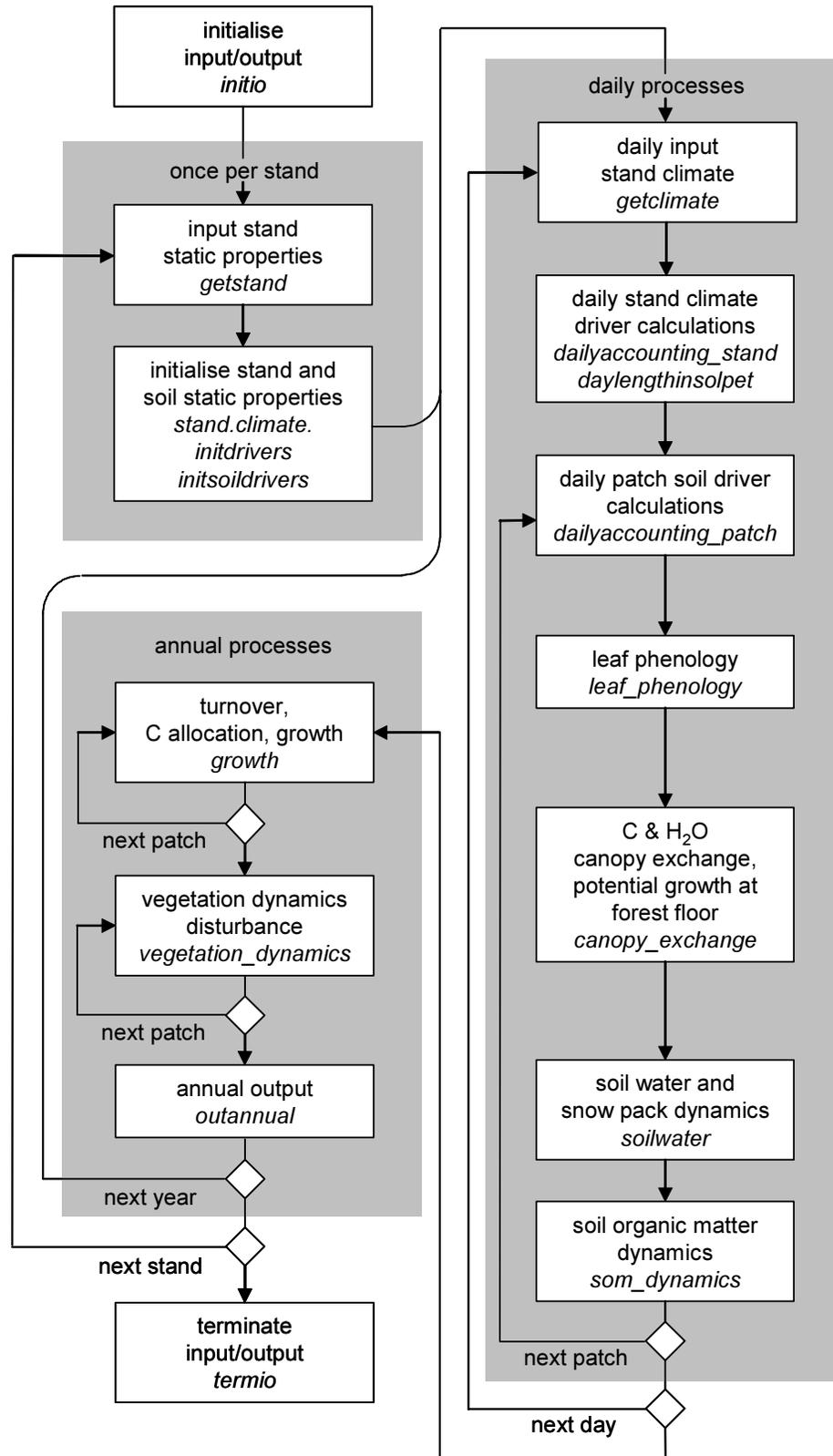


Figure 1. Flow structure of framework.

Modules

The modules, in addition to the framework, framework header file and binary libraries `gutil` and `plib`, are the components of computer code that make up the coupled model structure. The modules and their main functions are listed here.

Main module (main.cpp, [main.h]) Provides the interface between the model framework and the calling process (generally the operating system's command line; alternatively, another piece of software, such as the LPJ-GUESS windows shell).

Shell (shell.cpp, shell.h) Provides the global functions `dprintf`, `fail`, `plot` etc. which are available throughout the code of LPJ-GUESS. This family of functions is implemented by a subclass to the Shell class. By choosing which subclass to use (typically chosen in `main`, see above), the behavior can be altered. For instance, if the model is coupled to another model we might wish to perform logging in a different way.

Parameters module (parameters.cpp, parameters.h) Reads simulation settings and PFT parameters from the instruction script (`ins` file).

Input module(s) Reads and, if required, preprocesses climate and soil data used to drive the model. *It is the user's responsibility to provide code for a number of required input functions. Typically one of the included input modules will need to be modified for the data set which should be used, or a new input module needs to be written from scratch; see the section "Input/output", below.*

Output modules handle output of results to files. LPJ-GUESS comes with a standard output module called `CommonOutput` (`commonoutput.cpp`, `commonoutput.h`) which includes output of several commonly used results.

Driver module (driver.cpp, driver.h) Contains various functions for "preprocessing" environmental driver data to produce values for the driving parameters actually used by the process modules of the coupled model (e.g. calculates potential evapotranspiration, photosynthetically active radiation and daylength, given latitude and percentage of full sunshine; calculates soil temperature given air temperature and soil moisture status; interpolates monthly climate means to quasi-daily values); maintains records of climate (e.g. mean temperature for the last month; minimum coldest-month temperature for the last 20 years); updates various climate, soil and PFT state variables. Most functions are called from framework or input/output module.

Canopy exchange module (canexch.cpp, canexch.h, q10.h) Daily calculations of canopy-atmosphere exchange of H_2O and CO_2 ; AET (actual evapotranspiration); vegetation C-assimilation, vegetation N demand, autotrophic respiration, NPP, FPAR (fraction of incoming photosynthetically-active radiation used in photosynthesis), potential productivity at forest floor. NPP and AET calculations may be performed in diurnal mode, when appropriate forcing data are available (temperature and short-wave radiation).

Soil water module (soilwater.cpp, soilwater.h) Daily update of soil moisture status and snowpack size; runoff.

Soil organic matter dynamics module (sodynam.cpp, sodynam.h) Daily calculations of heterotrophic respiration, N mineralization, N fluxes and update of litter and SOM (soil organic matter) pool sizes.

Growth module (growth.cpp, growth.h) Daily update of leaf phenology; annual litter production and tissue turnover; annual allocation of assimilated C to plant tissues and reproduction; annual update of individual allometry (size, crown area etc).

Vegetation dynamics (vegodynam.cpp, vegodynam.h) Annual population dynamics (establishment and mortality), including introduction of new PFTs, disturbance by fire, generic patch-destroying disturbances in cohort/individual mode.

Emissions of biogenic volatile organic compounds (bvoc.cpp, bvoc.h, q10.h) Initialisation of standard emission rates, calculation of isoprene and monoterpene emissions as a function of plant photosynthesis and climate.

Libraries

Two binary libraries are currently required to build LPJ-GUESS. The libraries provide functions and classes of a purely technical nature which primarily serve to simplify the code of LPJ-GUESS itself. The functionality from the libraries most likely to be accessed by users and developers of the model code are described below in the sections "LPJ-GUESS classes" and "Input/output". Additional information is given as commenting in the library header files, plib.h and gutil.h.

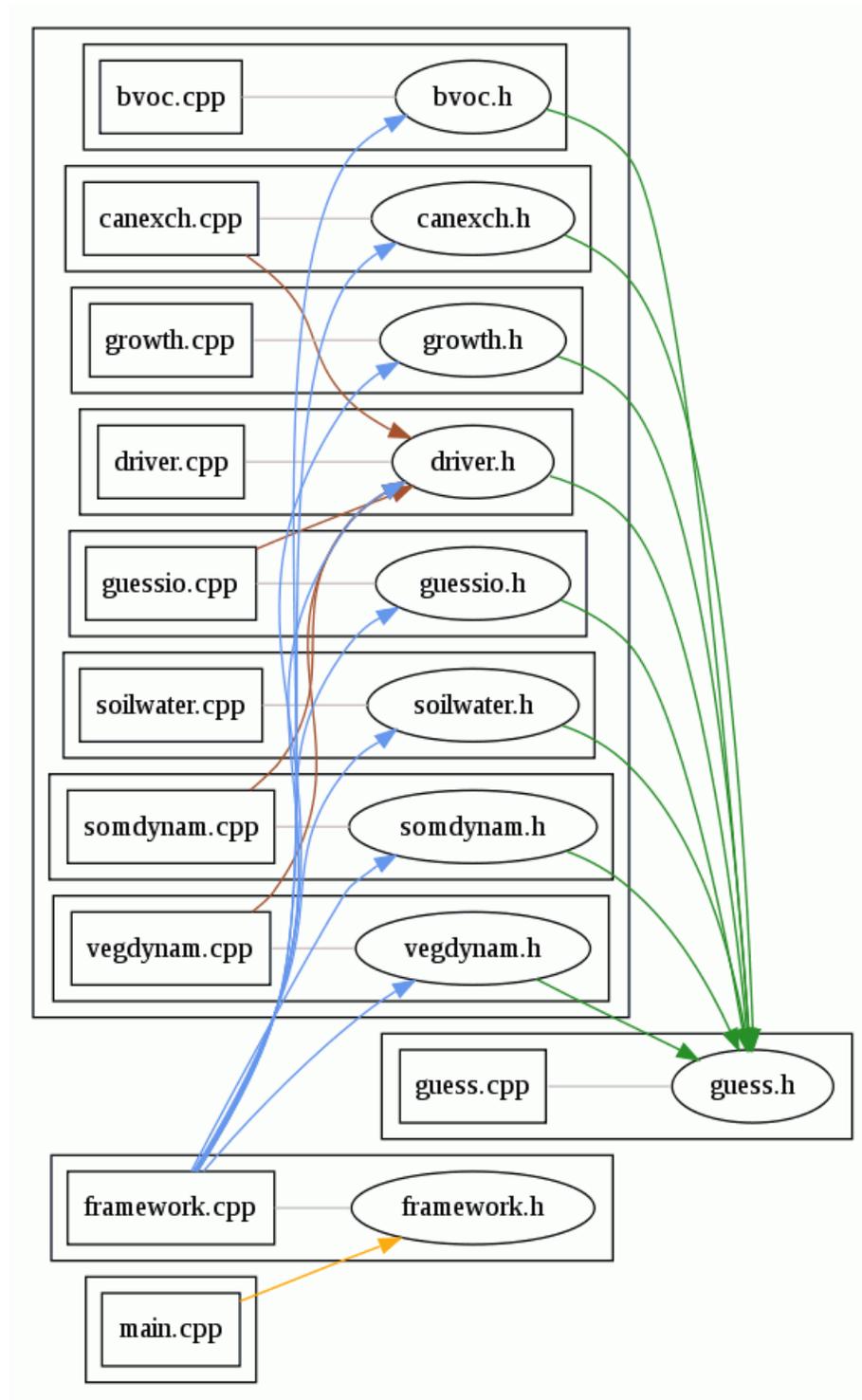


Figure 2. File structure of LPJ-GUESS. The file structure of the model reflects its modular design. Each module has its own source code file (box) and header file (ellipse). The arrows show dependencies. The framework function (in framework.cpp) calls each module, and all modules depend on guess.h where all the main classes are defined. The main module has a purely technical function – providing an interface between the model code and the platform (operating system etc) on which it is run. Many files and dependencies have been removed to simplify the picture, but this figure illustrates the general architecture.

LPJ-GUESS CLASSES

This section deals with the classes defined in the framework header file for LPJ-GUESS (usually called `guess.h`). These classes are the main medium by which data (environmental drivers, ecosystem state variables, PFT static parameters etc) are exchanged between modules and managed by the framework. Special operations are defined for some classes, and the use of these is also explained. Unavoidably, some parts of this section assume familiarity with some technical aspects of the C++ language; "C++ help" boxes are included to provide some help with these concepts.

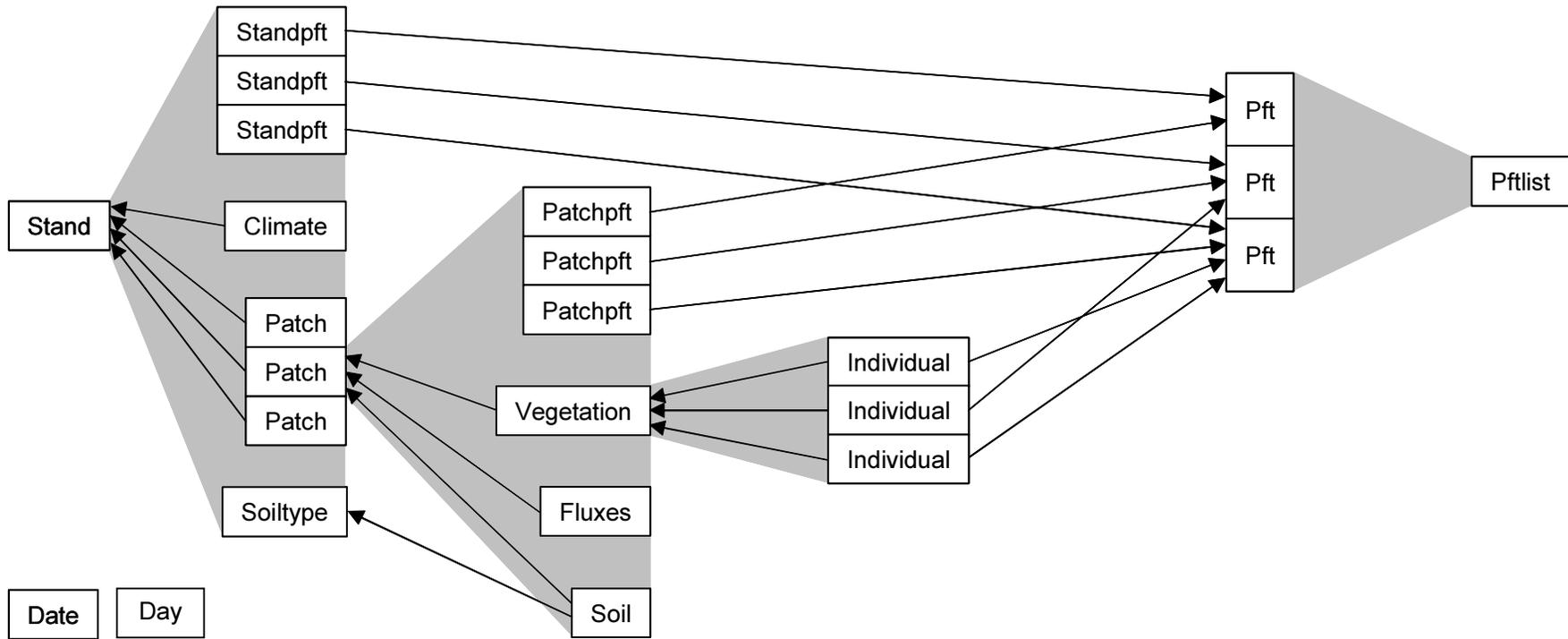


Figure 3. Class objects in LPJ-GUESS and relationships between them. Grey shading indicates membership of objects within a "parent" class (for example, objects of classes Standpft, Climate and Patch are members of class Stand). Arrows represent reference members: the class *from* which the arrow points contains a reference member variable to the (parent) object *to* which it points (for example, each Climate object contains a reference member variable to the parent Stand object).

Class objects defined in LPJ-GUESS

LPJ-GUESS uses, in a limited way, object-oriented features of the C++ language. All state and driver variables and nearly all static parameters are collected within classes. Classes are compound data structures which can contain both data (simple variables, arrays or other classes) and functionality.

For each modelled locality, there is a single object of the class Gridcell, containing all dynamic and static information specific to that locality, including the climate, soil and vegetation. If only natural vegetation is simulated, there is one object of the class Stand. If other landcover types (e.g. cropland, peatland) are simulated, each landcover is represented by one or more stands. Subsidiary to the stand is the patch, of which there may be one (population mode) or several (cohort and individual modes) in each stand for natural vegetation. Other landcover stands may have any other number of patches. Patches are represented by an object of class Patch which contains information about the soil, snow pack if there is one, and vegetation. Objects of class Individual contain information describing the average status of individual plants and are therefore subsidiary to the patch. Static parameters for plant functional types (PFTs) are also organised into a class, Pft, objects of which are stored as part of a collection class, Pftlist.

The globally-defined classes used by LPJ-GUESS and relationships between them are illustrated in Figure 3. A detailed description of each class is given below. Even more detail is provided by documentation in the framework header file, guess.h, where the classes are defined.

Object of class Gridcell

Top-level object containing all dynamic and static data for a particular gridcell (modelled locality). Member data include an object of type Climate, an object of type Soiltype, a list of Stand objects, and a list of Gridcellpft objects. Class Gridcell inherits functionality from the collection class template ListArray_idin3 in the gutil library (templates defined in gutil.h). Use syntax like the following to loop through and retrieve member Stand objects:

```
// Assume gridcell is an object of
// class Gridcell

gridcell.firstobj();
while (gridcell.isobj) {

    // Retrieve stand as a
    // linked list element
    Stand& stand = gridcell.getobj();
```

C++ Help: Classes

Classes are compound data types which may contain both data and functionality. The data contained within a class may be simple variables, arrays, or objects of other class types. Each data object is called a member variable. Classes may also contain member functions. Typically, but not necessarily, these perform operations using or modifying the member variables. Both member variables and member functions are accessed using the class member ('.') operator. For example, the code `ben.height` accesses the height member variable of a class object called ben; the code `ben.volume()` calls the object's member function volume.

C++ Help: Collection classes

A collection class is a class (compound data structure) that includes member functions for managing lists, arrays or other 'collections' of data, usually of a particular type. LPJ-GUESS uses collection classes called 'list arrays' which inherit their functionality from templates defined in the Gutil library (one of the custom libraries required to compile LPJ-GUESS). The use of templates means that different list array classes can accommodate different types of data. For example, classes Stand and Vegetation are both collection classes which inherit functionality from the template class ListArray_idin2, but class Stand 'contains' objects of class Patch, whereas Vegetation contains objects of class Individual. The list array types defined in Gutil provide functions for: (1) initialising or clearing the collection of objects; (2) adding a new object to the collection; (3) removing an object from the collection; (4) iterating (stepping) through the collection in sequential order; (5) accessing objects in the collection by index, i.e. as 'array' elements. Objects in the collection can be queried or modified directly – there is no need to create a new copy in memory.

```

        // NB: '&' is necessary
        // if modifying patch

    // Query or modify patch here

    gridcell.nextobj();
}

```

Object of class **Gridcellpft**

Object containing data common to *all* individuals of a particular plant functional type (PFT) in a particular gridcell (modelled locality). Used only in cohort and individual modes. State variables for the *average individual* of a PFT population in population mode are part of class Individual, not Gridcellpft. PFT static properties ("PFT parameters") are stored in objects of class Pft, not Gridcellpft. Gridcellpft objects are stored within a dynamic collection class object called pft which is a member of class Gridcell (see above). The Gridcellpft corresponding to a particular PFT in a particular stand can be accessed using the id (id code) member of the associated Pft object, and array-like syntax, similar to the following:

```

// Assume gridcell is an object of
// class Gridcell
// pft is an object of class Pft

Gridcellpft& gridcellpft =gridcell.pft[pft.id];
    // NB: '&' is necessary
    // if modifying gridcellpft

// Query or modify gridcellpft here

```

Object of class **Climate**

Contains all static and dynamic data relating to the overall environmental properties, other than soil properties, of a gridcell (modelled locality), i.e. temperature, precipitation, radiation, atmospheric CO₂ concentration, N deposition, derived parameters such as heat sums (growing degree-days, GDD), as well as latitude and day length. Soil static parameters are stored in an object of another class, Soiltype, which is also subsidiary to Gridcell; soil dynamic properties may vary for different patches and form part of a different class, Soil, subsidiary to the patch (see below). Class Climate includes a reference member variable which refers back to the Gridcell object of which the Climate object is a member. The parent Gridcell object can therefore be accessed using syntax such as the following:

```

// Assume climate is an object of class Climate

Gridcell& gridcell=climate.gridcell; // NB: '&' is necessary if modifying
stand

// Query or modify gridcell here

```

Object of class **Soiltype**

Stores soil static parameters. One object of class Soiltype is defined for each gridcell. Soil dynamic properties form part of class Soil, objects of which are subsidiary to the patch (see below).

Object of class **Stand**

Object containing all dynamic and static data for a particular stand (modelled locality, or grid cell). Member data include a list of Patch objects, and a list of Standpft objects. Class Stand inherits functionality from the collection class template ListArray_idin3 in the gutil library (templates defined in gutil.h). Use syntax like the following to loop through and retrieve member Patch objects:

```
// Assume stand is an object of
// class Stand

// Either:

for (p=0; p<stand.nobj; p++) {

    // Retrieve patch as an
    // array element
    Patch& patch=stand[p];
        // NB: '&' is necessary
        // if modifying patch

    // Query or modify patch here
}

// Or:

stand.firstobj();
while (stand.isobj) {

    // Retrieve patch as a
    // linked list element
    Patch& patch=stand.getobj();
        // NB: '&' is necessary
        // if modifying patch

    // Query or modify patch here

    stand.nextobj();
}
}
```

Object of class **Standpft**

Object containing data common to *all* individuals of a particular plant functional type (PFT) in a particular stand. Used only in cohort and individual modes. State variables for the *average individual* of a PFT population in population mode are part of class Individual, not Standpft. PFT static properties ("PFT parameters") are stored in objects of class Pft, not Standpft. Standpft objects are stored within a dynamic collection class object called pft which is a member of class Stand (see above). The Standpft corresponding to a particular PFT in a particular stand can be accessed using the id

C++ Help: References

A reference is an alternative name for an object (such as a variable or class object) residing somewhere in memory, by which the object can be accessed and queried or modified. A reference is similar in effect to a pointer (a variable holding the address of the object in memory), but avoids the need for sometimes confusing pointer syntax. Reference variables must be initialised when declared and, unlike pointers, can not subsequently be made to refer to a new object. Once declared and initialised, the reference name can be used exactly as if it were the 'real' name of the object it refers to. The symbol '&' after the type name in a variable (or class) declaration indicates that a variable of a reference type is being declared. For example, the following code declares a reference variable called result as an alternative name for a variable called value.

```
int value;
int& result=value;
```

Several LPJ-GUESS classes include one or more reference member variables referring to an object of another class. For example, class Vegetation includes a reference member called patch which refers to the parent Patch object. The following line of code retrieves the Patch object associated with a Vegetation object called vegetation and assigns the value 0.5 to the member variable fpar_grass of the Patch object:

```
vegetation.patch.fpar_grass=0.5;
```

Functions may also have a reference type. This means that they return a reference to an object residing somewhere in memory, rather than a copy of the object. For example, the various 'list array' classes (Stand, Vegetation, Pftlist) used in LPJ-GUESS all include the member function getobj, which returns a reference to the 'current' object in the collection of objects maintained by the class. The following line of code retrieves the current Pft object from a collection maintained by an object of class Pftlist and assigns the value 250.0 to the member variable longevity of the Pft object:

```
pftlist.getobj().longevity=250.0;
```

In LPJ-GUESS reference types are used in preference to pointers wherever possible. This improves readability of the model code and makes much of the internal functioning of the collection classes invisible to the 'user'.

(id code) member of the associated Pft object, and array-like syntax, similar to the following:

```
// Assume stand is an object of
// class Stand
// pft is an object of class Pft

Standpft& standpft=stand.pft[pft.id];
    // NB: '&' is necessary
    // if modifying standpft

// Query or modify standpft here
```

Object of class **Patch**

Stores data specific to a patch. In cohort and individual modes, replicate patches are required in each stand to accommodate stochastic variation across the site. In population mode, each stand contains just one subsidiary patch, representing average conditions for the entire stand. However, class Patch could be easily extended to represent within-stand heterogeneity, for example, with respect to soil properties. Member data include objects of type Vegetation, Soil, Fluxes and a list of Patchpft objects. Class Patch includes a reference member variable which refers back to the Stand object of which the Patch object is a member. The parent Stand object can therefore be accessed using syntax such as the following:

```
// Assume patch is an object of class Patch

Stand& stand=patch.stand; // NB: '&' is necessary if modifying stand

// Query or modify stand here
```

Object of class **Patchpft**

Object containing data common to *all* individuals of a particular plant functional type (PFT) in a particular patch, including litter pools. State variables for the *average individual* of a PFT population in population mode are part of class Individual, not Patchpft. PFT static properties ("PFT parameters") are stored in objects of class Pft, not Patchpft. Patchpft objects are stored within a dynamic collection class object called pft which is a member of class Patch (see above). The Patchpft corresponding to a particular patch can be accessed using the id (id code) member of the associated Pft object, and array-like syntax, similar to the following:

```
// Assume patch is an object of class Patch
// pft is an object of class Pft

Patchpft& patchpft=patch.pft[pft.id];
    // NB: '&' is necessary if modifying patchpft

// Query or modify patchpft here
```

Object of class **Vegetation**

A dynamic list of Individual objects (see below), representing the vegetation of a particular patch. Class Vegetation inherits functionality from the collection class template ListArray_idin2 in the gutil library (templates defined in gutil.h). Use syntax like the following to loop through and retrieve member Individual objects:

```
// Assume vegetation is an object of class Vegetation
vegetation.firstobj();
while (vegetation.isobj) {

    // Retrieve Individual as a linked list element
    Individual& indiv=vegetation.getobj();
    // NB: '&' is necessary if modifying patch

    // Query or modify patch here

    vegetation.nextobj();
}

```

New Individual objects (corresponding to a PFT population, cohort or individual plant; see class Individual, below) can be added to the dynamic list using the createobj member function of class Vegetation. The Pft objects associated with the new Individual object, and also the Vegetation object itself, must be specified as arguments to createobj, i.e.:

```
// Assume pft is an object of class Pft
vegetation.createobj(pft,vegetation);

```

An Individual object may be removed from the dynamic list using the killobj member function. The internal object pointer must first be set to point to the Individual object to remove. For example, the following code loops through all Individual objects in the dynamic list, removing those for which the condition survive(climate,indiv.pft) returns false:

```
vegetation.firstobj();
while (vegetation.isobj) {
    Individual& indiv=vegetation.getobj();
    if (!survive(climate,indiv.pft)) vegetation.killobj();
    else vegetation.nextobj(); // advance pointer only if indiv not killed
}

```

Class Vegetation includes a reference member variable which refers back to the Patch object of which the Vegetation object is a member. The parent Patch object can therefore be accessed using syntax such as the following:

```
Patch& patch=vegetation.patch; // NB: '&' is necessary if modifying patch

// Query or modify patch here

```

Object of class **Soil**

Stores state variables for soils and the snow pack. One object of class Soil is defined for each patch. Class Soil could easily be extended to include other environmental properties varying at spatial scales smaller than the stand (modelled locality, or grid cell). Class Soil includes a reference member variable which refers back to the Patch object of which the Soil object is a member. The parent Patch object can therefore be accessed using syntax such as the following:

```
// Assume soil is an object of class Soil
Patch& patch=soil.patch; // NB: '&' is necessary if modifying patch
// Query or modify patch here
```

Soil also includes a reference member variable referring to the Soiltype object containing static parameter values for this soil. Static parameters for this soil can therefore be accessed as in the following examples:

```
k=((soil.soiltype.thermdiff_15-soil.soiltype.thermdiff_0)/0.15*
    soilwater+soil.soiltype.thermdiff_0)*DIFFUS_CONV;

perc=soil.soiltype.perc_base*pow(wcont[i-1],soil.soiltype.perc_exp);
```

Object of class **Fluxes**

The Fluxes class stores accumulated monthly and annual fluxes. At present only fluxes of carbon and nitrogen are defined. Fluxes from ecosystems to the atmosphere are represented by positive values, fluxes from the atmosphere to ecosystems as negative values. One object of type Fluxes is defined for each patch. When fluxes are generated in the model, they can be reported to the Fluxes class with the report_flux function, e.g.:

```
// Report soil respiration
patch.fluxes.report_flux(Fluxes::SOILC, soil_respiration);
```

For fluxes associated with an Individual, a help function in the Individual class should be used instead:

```
// Report NPP for this individual
indiv.report_flux(Fluxes::NPP, ind_npp);
```

This makes sure we don't count fluxes for Individuals which are not yet "alive" (first year after establishment).

At the beginning of each simulation year, the Fluxes object should be reset (sets all fluxes to zero), by calling the reset function.

At the end of the year, the Fluxes object can be queried in order to print out annual or monthly fluxes. For PFT specific fluxes, values can be retrieved per PFT or as a patch total.

Object of class **Individual**

Stores state variables for an average individual plant. In population mode, this is the average individual for the entire 'population' of plants of a particular functional type (PFT) over the modelled area. In cohort mode, it is the average individual of a cohort of plants approximately

the same age and from the same patch. In individual mode, it corresponds to an actual individual in a particular patch (there is no averaging among similar individuals). Grass PFTs, however, are represented by a single average individuals in each patch, regardless of mode. Class Individual includes the reference member variable pft, which refers to the Pft object (see below) containing static parameters for the PFT to which the individual belongs. This Pft object can be accessed using syntax such as the following:

```
// Assume indiv is an object of class Individual
Pft pft=indiv.pft;
// Query pft here
```

Class Individual also includes a reference member variable which refers back to the Vegetation object in which the Individual object is stored. Class Vegetation, in turn, includes a reference to the parent Patch object (see classes Vegetation and Patch, above). The Patch object can therefore be accessed using syntax such as the following:

```
Patch& patch=indiv.vegetation.patch;
    // NB: '&' is necessary if modifying patch
// Query or modify patch here
```

See class Vegetation (above) for details of how to add and remove Individual objects from the dynamic list in which they are stored.

Object of class **Pftlist**

A dynamic list of Pft objects (see below). In general, there should be just one Pftlist object, containing static parameters for all possible plant functional types (PFTs). The static parameters for a particular PFT are accessed by objects of class Individual, Patchpft and Standpft via their pft reference member variable (see descriptions of these classes, above). Class Pftlist inherits functionality from the collection class template ListArray_id in the gutil library (templates defined in gutil.h). Use syntax like the following to loop through and retrieve member Pft objects:

```
// Assume pftlist is an object of class Pftlist
pftlist.firstobj();
while (pftlist.isobj) {
    // Retrieve Pft as a linked list element
    Pft& pft=pftlist.getobj();
        // NB: '&' is necessary if modifying pft
    // Query or modify pft here
    pftlist.nextobj();
}
```

Pft objects in the dynamic list may also be accessed using array-like syntax. The following code accesses the Pft object with id code (value of id member variable) 5:

```
Pft& pft=pftlist[5];
```

Object of class **Pft**

Stores static parameters for a plant functional type (PFT). Pft objects are stored within the dynamic list maintained by the (one and only) object of type Pftlist (see above). There should be just one Pft object for each PFT. Different average individuals of the same PFT (for example, representing different annual cohorts, or occupying different patches) access PFT static parameters via their pft reference member variable, which refers ("points to") the same Pft object in the PFT list.

Object of class **Date**

Class Date is a general-purpose class providing data and functionality for handling simulation virtual time. In general, the framework module should maintain a single object of type Date for stepping through simulation time (on a time step of one day). This object should be accessible from all modules (declared as an 'extern' object in the framework header file). Date contains the following member functions and variables. Note that variables should be *queried* only, never modified (except indirectly by calls to member functions init or next):

void **init**(int *nyearsim*)

Called to initialise timing to day 0 of year 0. Parameter *nyearsim* may be set to the number of years to simulate (then member variable *islastyear* will be set to true in the last year of the simulation).

void **next**()

Called at end of each simulation day to update information stored in member variables.

int **prevmonth**()

Returns the number (0=January, ..., 11=December) of the previous simulation month.

int **nextmonth**()

Returns the number of the next simulation month.

int **day**

The current Julian day of the year (0=1 Jan; 1=2 Jan; ...; 364=31 Dec).

int **dayofmonth**

The current day of the current month (0-27, 0-29 or 0-30 depending on month).

int **month**

The number of the current month (0=January, ..., 11=December).

int **year**

The current simulation year (0=first year).

bool **islastyear**

Value is true if this is the last year of the simulation, otherwise false.

bool **islastmonth**

Value is true if this is the last month *of the year*, otherwise false.

bool **islastday**

Value is true if this is the last day *of the month*, otherwise false.

bool **ismidday**

Value is true if this is the middle day of the month, otherwise false.

int **ndaymonth**[12]

The number of days in each month (e.g. 31, 28, 31, 30, ..., 31).

Object of class **Day**

Class `Day` is used to iterate over sub-daily periods. It is used within `canopy_exchange` function. Constructor initiates beginning-of-the-day period. In daily mode, this is the only period (which is also end-of-the-day period).

bool **isstart**, **isend**

Flags for the beginning and end of the day (in daily mode both flags are *true*)

int **period**

index (ordinal number) of the period within the day changes from 0 up to `date.subdaily`

void **next()**

move to the next sub-daily period. This method doesn't have a built-in stop condition; therefore, one needs to check *period* against `date.subdaily` before continuing iteration.

See, `canopy_exchange` for code sample.

Serialization

To convert all internal state in the form of objects and relationships to a flat form suitable for storage on file is called serialization. All classes which store part of the model's internal state (such as `Stand`, `Soil` or `Individual`) inherit from the base class `Serializable`, and must therefore implement the member function `serialize()`. The responsibility of the `serialize` function is to save or load all pertinent state information contained within the class to file. All variables needed for the program to resume execution later on should be included in the serialization.

The `serialize()` function takes an `ArchiveStream` as its only parameter. The `ArchiveStream` will either save or load data from file, and takes care of all technicalities of file I/O. In many cases, all the `serialize()` function needs to do is tell the `ArchiveStream` which member variables should be included.

Member variables of standard C++ primitive data types (such as `char`, `double` or `int`), can be serialized easily with the `&` operator. As an example, here's a small `Rectangle` class:

```
class Rectangle : public Serializable {
public:

    void serialize(ArchiveStream& arch) {
        arch & width & height;
    }

private:
    int width;
    int height;
};
```

By inheriting from `Serializable`, a class shows that it can be serialized. All serializable classes can also be serialized with the `&` operator:

```
class Book : public Serializable {
public:

    void serialize(ArchiveStream& arch) {
        arch & pages & dimensions;
    }

private:
```

```
    int pages;  
    Rectangle dimensions;  
};
```

In some cases it's necessary to do different things when saving and loading. For instance, the class might need to allocate memory for an object before it can be read from file. In these cases, the `save()` function in `ArchiveStream` can be queried, which returns true when writing to file and false when reading.

When running in parallel LPJ-GUESS will save state files from all processes in a single directory. Each process will create a file called `x.state`, where `x` is an integer identifying the process. Even though the files are organized like this, it is possible to resume execution with a different number of processes.

The standard version of LPJ-GUESS can save its state, or resume execution based on state files in a given directory. By default the state files are saved just after spinup, and execution is also resumed just after spinup when loading from state file. See the instruction file parameters `state_path`, `restart`, `save_state` and `state_year` for more information.

INSTRUCTION FILE

This section describes the instruction script (or “ins file”) used to configure the model. The ins file is a plain text file that contains (1) simulation settings, such as the number of years to run each simulation for, whether to model certain processes stochastically or deterministically, whether to run the model in population, cohort or individual mode; (2) PFT static parameters; and (3) "custom" information as required by a particular version of the input/output module, for example, the directories in which environmental data files are found.

Format

The instruction file used by LPJ-GUESS consists of commands, PFTs, groups, and comments, as described below. There are also so called “custom parameters”, which are a lightweight method for user supplied code to add their own parameters.

Commands

A command is a statement beginning with a keyword (typically the name of a setting) followed by an optional parameter value (a string, real or integer value). Some settings can also be given as a list of values. For instance:

```
title "Kiruna"
mtemp -14.2 -13.5 -9.5 -3.5 2.9 9.3 12.6 10.3 4.8 -1.5 -8.1 -12.1
include 1
```

A list of the keywords recognised (and, in most cases, required) in the ins file of LPJ-GUESS can be obtained by running the model with the command-line argument "-help". The keywords are also listed in an appendix to this reference.

PFTs

Plant functional types are created and configured in the ins file with the pft keyword, followed by the name of the PFT, and then the settings for that PFT enclosed in parenthesis. For instance:

```
pft "BNE" (
  leaflong 3
  turnover_leaf 0.33
  << etc. >>
)
```

Groups

Usually many PFTs will share some properties. To avoid duplicating those settings in each PFT, LPJ-GUESS supports groups of commands. A group is a named list of commands, which are inserted in the script when that name is used. For instance:

```
group "shade_tolerant" (
  est_max 0.05
  parff_min 350000
  << etc. >>
)

pft "BNE" (
  shade_tolerant
)
```

Comments

Comments start with an exclamation mark (!). They are ignored to the end of the line they appear on. For instance:

```
nyear_spinup 500 ! number of years to spin up the simulation for
```

Custom parameters

Custom parameters are a lightweight method for user supplied code to add their own parameters. The parameter values can be accessed easily in the code, and there's no need to define the parameters before they are used. The disadvantage of using them is that they don't show up in the help text (when the program is started with the `-help` argument), and the only error handling is to terminate the program if a requested parameter is missing from the ins file.

The custom parameters may be included in the ins file using syntax similar to the following examples:

```
param "co2" (num 340)
param "file_gridlist" (str "gridlist.txt")
```

The first example above specifies a numerical value (340, or 340.0) for a parameter called "co2"; the second example specifies a character string ("gridlist.txt") for a parameter called "file_gridlist".

When the ins file is read, all "param" settings, like the ones above, are stored by an instance of class Param (with a capital "P") called param (small "p"). The object param is accessible anywhere within the LPJ-GUESS source code.

The values associated with the "param" strings in the above examples can be retrieved using the following function calls (remember that *the ins file must have been read in first*):

```
param["co2"].num
param["file_gridlist"].str
```

Each "param" item can store *either* a number (int or double) *or* a string, but not both types of data. The global function **fail** is called to terminate output if a "param" item with the specified identifier was not read in.

Using multiple files

An ins file can include another ins file with the special "import" keyword:

```
import "guess.ins"
```

LPJ-GUESS will then read in and parse the imported ins file before continuing. If a relative path name is used, it will be relative to the current (importing) ins file (not relative to the directory where the model was started!). This feature is used to structure ins files in a way such that duplication can be reduced. The importing ins file will typically start by importing one or several files, and then adding its own settings or overriding some of the settings from

the imported files. If one setting occurs in several places, the last one is the one which will ultimately be used by the model.

Parameters module

The parameters module (parameters.h and parameters.cpp) is responsible for reading in and understanding the ins file. This is where all parameters are defined. A parameter is defined by specifying its name, where to store its value, its data type and expected range of values, and a short documentation string describing its meaning for the user.

As described above, custom parameters can be used in user supplied code. In that case, there is no need to modify the parameters module. The parameters module also exposes a small interface for other modules to define “proper” parameters. This must then be done before the instruction file is read in by the parameters module. Details about this are given in the module’s header file, parameters.h.

Most of the model’s standard parameters are defined within the parameters module itself. LPJ-GUESS uses functionality from the "plib" library (which is required to build an executable of the model) to read settings from the ins file. Details of plib functionality are given in the library's header file, plib.h, which should be consulted if adding new general settings for inclusion in the ins file.

INPUT

This section describes the input module(s) of LPJ-GUESS, through which all driver data are fed to the model framework. **Users of LPJ-GUESS will normally have to write part of the code of the input module themselves, or create a new input module, to suit their particular application and driver data (file formats, etc).** This part explains how to perform such coding.

The input module(s) of LPJ-GUESS

In general, it is the responsibility of each user of the model to provide code for the input module. LPJ-GUESS is distributed with two input modules. One for the CRU data set (data set not included with the LPJ-GUESS distribution), and one for *demonstration purposes only* (small data set included). More information about these can be found in the source code documentation of the classes for each input module (CRUInput and DemoInput).

Creating an input module

Every input module is a subclass of the InputModule class. InputModule is a so called abstract base class; it doesn't contain any implementation itself, it just declares a number of member functions which the subclasses must implement. To create a new input module, create a new class which inherits from InputModule, implement all the member functions described below, and register it with the REGISTER_INPUT_MODULE macro (example below).

Required member functions

void **init()**

Initialises the input module (e.g. opening files). The framework will call this function after reading in the instruction file, so parameters from the instruction file (for instance path names to input files) can be used. The input module's constructor however will be called before the instruction file is read, so if parameters need to be declared they should be declared in the constructor, not in init().

bool **getgridcell**(Gridcell& *gridcell*)

Obtains coordinates and soil static parameters for the next gridcell (locality) to simulate. The function should return false if no gridcells remain to be simulated, otherwise true. Currently the following member variables of *gridcell* should be initialised: longitude, latitude and *climate.instype*; the following members of member *soiltype*: *awc*[0], *awc*[1], *perc_base*, *perc_exp*, *thermdiff_0*, *thermdiff_15*, *thermdiff_100*. The soil parameters can be set indirectly based on an lpj soil code (Sitch et al 2000) by a call to function **soilparameters** in the driver module (driver.cpp):

```
soilparameters(gridcell.soiltype, soilcode);
```

If the model is to be driven by quasi-daily values of the climate variables derived from monthly means, this function may be the appropriate place to perform the required interpolations. The utility functions **interp_monthly_means** and **interp_monthly_totals** in driver.cpp may be called for this purpose:

```
interp_monthly_means(mtemp, dtemp);
interp_monthly_totals(mprec, dprec);
```

Both functions take as input an array of 12 doubles, and give as output an array of 365 doubles. **interp_monthly_means** assumes the input values are monthly averages, **interp_monthly_totals** assumes they are monthly sums.

In diurnal mode, currently the only two valid values for *climate.instype* variable are *NETSWRAD* and *NETSWRAD_TS*. Two additional insolation types (*SWRAD* and *SWRAD_TS*) could be supported with minor modification to the **daylengthinsoleet** function (see comments in code).

```
bool getclimate(Gridcell& gridcell)
```

Called by the framework each simulation day to obtain climate data (including atmospheric CO₂ and insolation) for this day. The function should return false if the simulation is complete for this gridcell, otherwise true. This will normally require querying the *year* and *day* member variables of the global class object *date*, the simulation time step:

```
if (date.day==0 && date.year==nyear) return false;
// else
return true;
```

Currently the following member variables of the *climate* member of *gridcell* must be initialised: *co2*, *temp*, *prec*, *insol* and *dndep*. In the case of a simulation with BVOC, the additional parameter *dtr* needs to be set. If the model is to be driven by quasi-daily values of the climate variables derived from monthly means, this day's values will presumably be extracted from arrays containing the interpolated daily values (see **getgridcell**, above):

```
gridcell.climate.temp=dtemp[date.day];
gridcell.climate.prec=dprec[date.day];
gridcell.climate.insol=dsun[date.day];
gridcell.climate.dndep=dndep[date.day];

// bvoc
gridcell.climate.dtr=ddtr[date.day];
```

If the model is run in diurnal mode, which requires appropriate climate forcing data, additional members of the *climate* must be initialised: *temps*, *insols*. Both of the variables must be of type *std::vector*. The length of these vectors should be equal to value of *date.subdaily* which also needs to be set either in **getclimate** or **getgridcell** functions. *date.subdaily* is a number of sub-daily period in a single day. Irrespective of the BVOC settings, *climate.dtr* variable is not required in diurnal mode.

```
void getlandcover(Gridcell& gridcell)
```

Sets land cover fractions for the gridcell for the current year. The function should set the values of the *landcoverfrac* member of *gridcell*. *landcoverfrac* is an array of values describing the land cover fractions for each land cover type. Each value should be between 0 and 1, and the sum of the whole array should be 1.

An example input module

The following example illustrates how to create an input module from a technical perspective, with uninteresting forcing data. For more realistic examples, see the input modules included with LPJ-GUESS.

```
#include "inputmodule.h"

class MyInput : public InputModule {
public:
```

This first part starts the class definition and inherits from InputModule, which is found in inputmodule.h

```
MyInput()
    : first_gridcell(true) {
}

void init() {
}
```

We then define a constructor and implement the init function. In this case the constructor simply initialises a member variable that keeps track of when we've handled the first grid cell. The init function does nothing in this trivial example. In the example all functions are defined inline in the class definition to keep it simple, in real code the class definition and function bodies are usually separated into .h and .cpp files.

```
bool getgridcell(Gridcell& gridcell) {
    if (first_gridcell) {

        first_gridcell = false;

        gridcell.set_coordinates(-17.01, 32.74);
        gridcell.climate.instype = SUNSHINE;
        soilparameters(gridcell.soiltype, 2);

        return true;
    }
    else {
        return false;
    }
}
```

The getgridcell function in this example has hard coded values for a single location, and returns false the second time it is called to indicate that the simulation is finished.

```
bool getclimate(Gridcell& gridcell) {
    if (date.year < nyear_spinup + 100) {
        Climate& climate = gridcell.climate;

        climate.co2 = 300;
        climate.temp = 15;
        climate.prec = 2;
        climate.insol = 75;
        climate.dtr = 5;
        return true;
    }
    else {
        return false;
    }
}
```

```

    }
}

```

The `getclimate` function also simply returns hard coded values in this example. In a real input module, the date object would be used to decide which values to use. In this example the date object is only used to determine when to finish simulating the current grid cell (100 years after the spin up is complete).

```

void getlandcover(Gridcell& gridcell) {
    std::fill_n(gridcell.landcoverfrac, int(NLANDCOVERTYPES), 0.0);
    gridcell.landcoverfrac[NATURAL] = 1;
}

```

The `getlandcover` function in this case sets all fractions to zero, except the NATURAL land cover type.

```

private:
    bool first_gridcell;
};

```

Finally we complete the class by defining the member variable.

To make sure the new input module is used by the framework, it needs to be registered. The registration is done with a macro, normally in the module's `.cpp` file:

```

REGISTER_INPUT_MODULE("myinput", MyInput)

```

The first argument here is the name of the input module, and the second is the class. When the module is properly registered, it will be possible to choose it by specifying it in the command line parameters. In this case, by specifying “-input myinput”. If no input module is specified, the CRUInput module is used.

OUTPUT

This section describes the output module(s) of LPJ-GUESS, responsible for generating results from the model. LPJ-GUESS is distributed with an output module for many common outputs, but the user may need to supply code for additional outputs. This can be done either by modifying the standard output module, or by creating a new output module.

The output module(s) of LPJ-GUESS

The standard output module distributed with LPJ-GUESS is a class named `CommonOutput` (`commonoutput.h` and `commonoutput.cpp`). If additional outputs are needed, it can be modified, or an additional output module can be created.

Creating an output module

Every output module is a subclass of the `OutputModule` class. `OutputModule` is a so called abstract base class; it doesn't contain any implementation itself, it just declares a number of member functions which the subclasses must implement. To create a new output module, create a new class which inherits from `OutputModule`, implement all the member functions described below, and register it with the `REGISTER_OUTPUT_MODULE` macro (example below).

```
void outannual(Gridcell& gridcell)
```

Called by the framework at the end of the last day of each simulation year to provide the opportunity to output simulation results. This function does not have to provide any information to the framework.

```
void outdaily(Gridcell& gridcell)
```

Called by the framework at the end of each simulation day to provide the opportunity to output simulation results. This function does not have to provide any information to the framework.

The functions should generate results based on the information contained in the `gridcell` object. It's possible to generate the results in many different ways, such as creating and writing to text files. The recommended way however is to use the `OutputChannel` interface. Using an `OutputChannel` means that all the results from the model will be generated in the same way, your output module doesn't need to be concerned about file formats or text formatting, and you can change all outputs by changing the current `OutputChannel` if you want to have a different format. `OutputChannel` and its related classes are defined and documented in `outputchannel.h` and `outputchannel.cpp`. Examples of using it can be found in the `outannual` function in `CommonOutput`.

When the new `OutputModule` subclass is defined, it should be registered with the `REGISTER_OUTPUT_MODULE` macro, typically in the output module's `.cpp` file:

```
REGISTER_OUTPUT_MODULE("myoutput", MyOutput)
```

The first argument here is the name of the input module (currently not used), and the second is the class. Once the output module is registered, the framework will use it. All registered output modules are used, the order in which the framework calls their functions is undefined.

INPUT/OUTPUT TECHNIQUES

This section deals with input/output using "streams" in the C and C++ languages. It includes recommendations as to which functions from the standard C/C++ libraries and the custom libraries forming part of LPJ-GUESS should be used to perform input and output-related operations such as opening and closing files, and reading and writing data to and from files, the keyboard and the screen. There are also brief synopses and example code explaining how all the recommended functions may be used.

Recommended input/output techniques for use in LPJ-GUESS

The standard libraries of the C and C++ languages include several "families" of functions and classes for performing input from and output to files (and the keyboard, screen etc). Users of LPJ-GUESS with prior experience of C/C++ programming may have established preferences as to coding input/output, and are in practice free to use whatever functions they wish. However, there are arguments for adopting a standard subset of techniques:

- LPJ model development and application is a collaboration. Even "personalised" versions of the model code are subject to viewing, copying or application by other developers. Few LPJ developers are programmers in the first instance, and even fewer have a background working with the C or C++ languages. "Keep it simple" is surely a helpful principle.
- The functions adopted here, with one exception, are defined in the standard C runtime library and so are supported by all C and C++ compilers. This guarantees portability of any code built using these functions. The function **readfor**, recommended here for input of ASCII text data, is defined in the `gutil` library, which is required to compile LPJ-GUESS. The library itself is ANSI-compatible. This function does not, therefore, restrict portability of any code it is included in.

Input/output using streams

The functions described here (with the sole exception of **readfor**) are defined in the header file `stdio.h` and form part of the standard C runtime library, implemented by all C and C++ compilers. These functions use special memory areas called streams to store data temporarily before it is transferred to, or after it is retrieved from, a data file on disk. The streams are managed automatically by the library and are consequently "transparent" to the calling program.

Streams implemented by a calling program are identified by a "handle", which is a variable of the type `FILE*` ("pointer to FILE"). Whenever a stream is created for reading or writing (usually, by opening a file with the **fopen** function), the stream handle must be stored in a variable of type `FILE*` and "remembered" until the stream is closed (either by termination of the program or, preferably, by a call to function **fclose**). The handle is required every time data is sent to, or retrieved from, the stream. The handle is also used when performing operations on the "file pointer", such as "rewinding" the file.

A few streams are implemented automatically and are, by default, always open for reading or writing. The most useful of these are `stdin`, which by default retrieves input from the keyboard, and `stdout`, which sends output to the screen.

Required header files

Any program using the functions described here must include the header files `stdio.h` and `gutil.h`; i.e. the following `#include` directives must appear at or near the top of each source code file (note that, if both `#include`'s are present, they should appear in the same order as shown here):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <gutil.h>
    // NB: the angle bracket <...> notation assumes the path to
    // gutil.h is in the standard include path
    // (see compiler documentation)
```

The header files `gutil.h`, `stdlib.h`, `string.h` and `time.h` are required only if function **readfor** is called by the program. In this case, the program must also be linked to a binary form of the `gutil` library (which has the filename `gutil.lib` or `gutil.a`). Note that all the above `#include`'s appear in the default framework header file for LPJ-GUESS (`guess.h`); therefore, they do not also have to appear in the input/output module's source code file, so long as `guess.h` is included as a header file (as it always should).

Opening a file for input or output

Use the **fopen** function to open a file for reading or writing. The file must exist if it is to be opened for reading only. If opening a file for writing, a new file is created, or the file is overwritten if it already exists. This function returns a handle to the resultant input or output stream, which should be stored in a variable of type FILE* by the calling program. If the handle is null (has the value 0), this means that the specified file could not be opened.

Function synopsis:

FILE* **fopen**(const char* *filename*, const char* *mode*)

Arguments:

<i>filename</i>	a string identifying the file to open. The full pathname should be given unless the file is located in the "working directory" for the calling program – usually the directory in which the executable is stored. If the filename is specified as a character string literal, note that the backslash character ("\") must be specified by <i>two consecutive</i> backslash characters (this is because the backslash character has a special meaning in C/C++ string literals and a single backslash character is never defined); for example, use "c:\\guess\\output.txt"; not "c:\guess\output.txt"; of course, this rule applies only to string literals <i>within the source code</i> of a program.
mode	the type of stream to create; various modes are possible; the most useful ones are:
"r" or "rt"	open stream for input (reading) of ASCII text (the file must exist)
"w" or "wt"	open stream for output (writing) of ASCII text (a new file is created, or the existing file is opened and its contents erased)
"a+" or "a+t"	open stream for text output in append mode – any output is added to the end of the original file contents; a new file is created if the specified one does not exist.
"rb"	open stream for input of binary data (the file must exist)
"wb"	open stream for output of binary data (a new file is created, or the existing file is opened and its contents erased)
"a+b"	open stream for binary output in append mode (see "a+").

Note: arguments *filename* and *mode* are of type const char* (the standard C string type) by default; however, arguments of type xstring (the class type used for strings in LPJ-GUESS) can also be specified (they are casted automatically to char*).

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <gutil.h>

FILE *in,*out;
xstring logfile="c:\\guess\\guess.log";

// Open text file "temp.bin" for input
in=fopen("temp.bin","rb");
if (!in) {
    printf("Error: could not open temp.bin for input\n");
    exit(99);
}

// Open log file with stored name for output
out=fopen(logfile,"wt");
if (!out) {
    printf("Error: could not open %s for output\n", (char*)logfile);
    exit(99);
}
```

Input/output of binary data

Binary data can be efficiently written to a file using the **fwrite** function, and retrieved using the very similar **fread** function. Note that binary data are transferred byte-by-byte exactly as stored in memory. Different compilers (e.g. Sparc versus Microsoft) and, more importantly, different languages (e.g. C/C++ versus FORTRAN) may use different ways of storing the same information in memory. This applies particularly to numerical data types (integers and floating-point numbers). Therefore, binary files should be avoided as a means of data transfer between different systems and programs built using different compilers.

Function synopses:

```
size_t fread(void* buffer, size_t size, size_t count, FILE* stream)
size_t fwrite(const void* buffer, size_t size, size_t count, FILE* stream)
```

The returned value is the number of *items* (NB: not *bytes*) actually read or written; this should normally be the same value as argument *count*, but may be less if an error occurred or (for **fread**) the end of the file was reached before all items were input.

Arguments:

<i>buffer</i>	A pointer to (i.e. the address in memory of) the area in memory to write to (for fread) or transfer to file (for fwrite). The argument supplied is typically the name of an array (equivalent to the address in memory of the first element of the array) or the '&' operator followed by the name of a variable or class object.
<i>size</i>	The number of bytes in memory taken up by each value to transfer. Usually the sizeof operator is used to retrieve this value; for example <code>sizeof(int)</code> returns the number of bytes occupied by a single value of type <code>int</code> .
<i>count</i>	The number of consecutive objects of the specified <i>size</i> to transfer. This value is usually set to 1 if transferring a simple variable or class object, or, if transferring an array, the number of elements in the array.
<i>stream</i>	The stream to write to or from.

Note: the type `size_t` (size type) is defined in the header file `stdio.h`; it can be treated as an ordinary integral type (`int`, `long` etc).

Example:

```
int i,nitem=5;
double value[]={ 12.3, 4.6, 9.2, 0.7, 3.6 };
double newvalue[5];

// Open data.bin for output
FILE* out=fopen("data.bin","wb");

// Write data to stream out
fwrite(&nitem,sizeof(int),1,out);
fwrite(value,sizeof(double),nitem,out);

// Close data.bin
fclose(out);

// Reopen data.bin for input
FILE* in=fopen("data.bin","rb");

// Read data to array newvalue
fread(&nitem,sizeof(int),1,in);
fread(newvalue,sizeof(double),nitem,in);

// Close data.bin
fclose(in);
```

Input of ASCII text

The standard C runtime library provides function **fscanf** for general-purpose input of data in text format. However, **fscanf** has some limitations with regard to input of "fixed format" data as commonly used in FORTRAN programs. In addition, the specifier codes used to describe data formats for **fscanf** are idiosyncratic and will take some time for programmers accustomed to FORTRAN format strings to become used to. Therefore, for input of ASCII text data, a function from the *gutil* library (*gutil.lib* or *gutil.a*) is recommended in place of the standard C function. This function, **readfor**, employs format strings very similar to those used in FORTRAN. The format string argument of **readfor** is of type *xstring* (also defined within the *gutil* library), and strings input using **readfor** are also converted to *xstring* objects.

Function **readfor** is available by default in the input/output module of LPJ-GUESS (the *gutil* library is a required component for building LPJ-GUESS). Other programs using **readfor** must be linked to *gutil.lib/gutil.a*, and include the header file *gutil.h* (see "Required header files", above).

Function synopsis:

```
bool readfor(FILE*& strm, xstring format, ...)
```

Reads in ASCII text data from input stream *strm*, according to a FORTRAN-style format specification given in the string *format*. Addresses of the variables to be assigned to are listed, in order of assignment, in the ellipsis (...) argument list of the function. By default, any characters remaining on the current line are discarded when the statement terminates (this behaviour can be overridden by a \$ specifier in the format string – see below). The function returns true if all specified values could be read in and assigned, or false if an end-of-file condition prevented some values from being read in and assigned.

Example:

The following code opens for reading a text file called "climate.txt", reads in two values of type double, one integer, and 12 further values of type double, and assigns these values, respectively, to variables *lon*, *lat*, *elev*, and the 12 elements of array *mdata*. Note that the arguments following the format string are *addresses* of the variables to be assigned to – use the '&' prefix for simple variables; omit the '&' if specifying an array name (a pointer).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <gutil.h>

FILE* in;
double lon,lat;
int elev;
double mdata[12];

in=fopen("climate.txt","rt");
if (!readfor(in,"f6.1,f5.1,i4,12f5.1",&lon,&lat,&elev,mdata))
    printf("Warning: not all values could be read in\n");
```

Format specifiers

A subset of the format specifiers defined for input in the FORTRAN-77 language are supported by function readfor. Note that there are some differences in the way these are implemented compared to FORTRAN. Specifier fields in the format string are normally separated by commas. Each instance of an F, I or A specifier is assumed to correspond to one argument in the ellipsis argument list. Specifier syntax is case-insensitive and spaces and tabs in the format string are ignored.

Important: a value (other than 1) for n (number of items), if specified as part of an F, I or A specification, assumes the values read are to be assigned to consecutive elements of an array, whose starting address is given by a *single* argument in the argument list. Do not use this feature to assign multiple values with the same input format to multiple consecutive arguments.

Floating point specifier: $nFw.dEe$

Reads one or more floating point numbers, assigning each value to a variable of type double.

- n the number of items to read in; if n is not specified, one value is assumed. See cautionary note above.
- w the number of characters to read in for each item; if not specified, characters are read up to the next space, tab, end-of-line, or the next instance on the current line of a separator character appearing immediately after the specifier in the format string (see below).
- d the number of digits in the fractional part of the number; if a decimal point is encountered, this overrides the specified value. If d is omitted and no decimal point is encountered, the input string is interpreted as a whole number.
- e the number of digits in the exponent part of the number; if the character 'E' or 'e' is encountered, this overrides the specified value. If e is omitted, the exponent 0 (i.e., $10^0 = 1$) is assumed.

Examples:

Input stream: 1234 5.67

<i>format specification</i>	<i>strings read</i>	<i>values assigned</i>
F	"1234"	1234.0
F3	"123"	123.0
F3.1	"123"	12.3
F4.2E1	"1234"	1.23E+04
2F4.2	"1234", " 5.6"	12.34, 5.6

Integer specifier: *nIw*

Reads one or more integers, assigning each value to a variable of type int.

- n* the number of items to read in; if *n* is not specified, one value is assumed. See cautionary note above.
- w* the number of characters to read in for each item; if not specified, characters are read up to the next space, tab, end-of-line, or the next instance on the current line of a separator character appearing immediately after the specifier in the format string (see below).

Examples:

Input stream: 123 4567

<i>format specification</i>	<i>strings read</i>	<i>values assigned</i>
I	"123"	123
I2	"12"	12
2I4	"123 ", "4567"	123, 4567

Character string specifier: *nAw*

Reads one or more character strings, assigning each to a variable of type xtring.

- n* the number of items to read in; if *n* is not specified, one value is assumed. If reading multiple strings, a width (*w*) specification must be included (otherwise the rest of the current line is read in and assigned to the first xtring variable; subsequent variables are assigned null strings). See cautionary note above.
- w* the number of characters to read for each item; if not specified, characters are read up to the next whitespace character, comma, or the next instance of a separator character appearing immediately after the specifier in the format string (see below).

The read to end-of-line specifier # may be used to read to the end of the current line (see below).

Examples:

Input stream: BERT HIGGINS

<i>format specification</i>	<i>strings read and assigned</i>
A	"BERT"
A2	"BE"
2A6	"BERT H", "IGGINS"
A#	"BERT HIGGINS"

Position specifier: nX

Advances one or more characters on the input stream.

n The number of characters to read and discard; if n is omitted, a single character is read in and discarded.

End-of-line specifier: /

Advances to the end of the current line on the input stream. Input continues from the start of the next line.

Suppress line feed specifier: \$

If given as the last significant character in the format string, suppresses reading and discarding of the remainder of the current line on the input stream.

Read to end-of-line specifier: #

If specified after a character string specifier, reads to the end of the current line. Use to read in the rest of the current line as a single character string, including white space characters or commas which are otherwise interpreted as separator characters.

Separator character specifier:

Any character other than a space, tab or comma, that cannot be interpreted as part of one of the specifiers above, is interpreted as a separator character, and causes input up to and including the first instance of the specified separator character on the current line. If given immediately following a variable-width F, I or A specification, input continues, for each item, until an instance of the specified separator character is encountered, or the end of the current line is reached (otherwise input continues until a space, tab or the end of the line is reached). If the separator character follows a fixed-width F, I or A specification, or forms a separate specification field, the characters read are discarded.

A comma is interpreted as an *optional* separator character; a variable-width F or I specification followed by a comma results in input of text up to the next space, tab, end-of-line or comma, *whichever comes first*; a variable-width A specification followed by a comma results in input up to the next comma or end-of-line, whichever comes first.

Examples:

Input stream: 123; Hello World!; 3.142

<i>format specification</i>	<i>strings read</i>	<i>values assigned</i>
I;A;F	"123", " Hello World!", "3.142"	123, " Hello World!", 3.142
F2.1;;I2,A	"12", " 3", ".142"	1.2, 3, ".142"

Input from the keyboard

If input is required from the keyboard instead of a stream attached to a file, function **readfor** can be used specifying `stdin` (one of the default streams defined in `stdio.h`, and normally associated with the terminal keyboard) as the *strm* argument. Due to portability considerations, however, keyboard input should be avoided in LPJ-GUESS (run-time options may be set via custom specifiers in the instruction script [.ins] file

Output of ASCII text

Formatted output of text is achieved using the "printf" family of functions. LPJ-GUESS supplies its own member of this family, **dprintf**, which sends output to the screen log (if enabled) and log file (if enabled). In general, **dprintf** should be used instead of **printf** for "log" type output from LPJ-GUESS, including output to the screen. This improves portability (for example, between command-line and Windows-shell implementations of LPJ-GUESS) and allows screen output to be disabled if, for example, the model is to be set up to run in batch mode. LPJ-GUESS provides function **fail** to send text output (usually some form of error message) to the screen log and log file, then end the simulation.

Function synopsis:

```
void dprintf(const char* format, ...)
```

Intended for writing ASCII text data to the screen or screen log window, and/or a log file (by default, guess.log). The function may be called from anywhere within the code of LPJ-GUESS. The function is declared in shell.h, and defined in shell.cpp. The default destination for the output text can be changed by simple modification of the function definition in shell.cpp. In other respects, the function behaves exactly like function **fprintf**, which is described below.

Function synopsis:

```
void fail(const char* format, ...)
```

Writes ASCII text data to the screen or screen log window, and/or a log file (by default, guess.log), then terminates execution of the model. Intended to allow a final error message or similar to be output before aborting the model simulation, for example in the event of an abnormal condition suggesting that an error has occurred. The function may be called from anywhere within the code of LPJ-GUESS. The function is declared in shell.h, and defined in shell.cpp. The function is identical to **dprintf** in its handling of output.

For output to files, use the **fprintf** function. A simplified description of function **fprintf** follows; a complete synopsis of the function is beyond the scope of this guide (see any C language reference for a more detailed description of this function).

Function synopsis:

```
int fprintf(FILE* stream, const char* format, ...)
```

Writes ASCII text data to output stream *stream*, according to the format specifications and other text given in the string *format*. Arguments to be output are listed in the ellipsis (...) argument list of the function. A format specification is required for each argument in the ellipsis argument list. The arguments must appear in the same order as the format specifications in *format*. The returned value is the number of characters output.

The format string

The format string may consist of any combination of ordinary text and format specifications. Ordinary text is output exactly as it appears in the format string, while each format specification is converted to a representation of the data value contained in (or pointed to) by its corresponding argument in the ellipsis argument list. Format specifications may be freely interspersed with ordinary text.

Format specifications

Each format specification has the form: `%fwpt`

where *f*, *w*, *p* and *t* are different fields of the specification, known as the flags field, width field, precision field and type field, respectively. All of the fields are optional (may be omitted), except the type field.

The flags field, *f*

The flags field, if given, may consist of any combination of the following single character specifiers:

- left align the output text within field specified by *w* (default: right align)
- + if the value to output is of a signed numerical type, prefix the output text by a sign (+ or -) (default: no sign output)
- 0 if the value to output is of a floating point type, pad the output value on the left by zeroes, up to the width specified by *w* (default: padding by spaces)

The width field, *w*

If given, this field must consist of a positive integer, representing the *minimum* number of characters to output. If the output value requires more than the number of characters specified by *w* (depending on the precision field, the flags field and the value of the argument to be output) the width specification is ignored. If the number of characters to output is *less* than *w*, the output is padded on the left by spaces, by default. This default behaviour can be modified by a 0 or – specifier in the flags field (see above).

The precision field, *p*

This field always consists of a decimal point (.) followed by a positive integer, representing *either* the number of digits to show after the decimal point for floating point numbers output using the e, E or f type specifier (see below); *or* the maximum number of significant digits to print for floating point numbers output using the g or G type specifier (see below). Default precision is assumed if the precision field is omitted. Precision specifications can also be made for output of integer and string types, but this functionality is not explained here.

The type field, *t*

The type field specifies the data type of the corresponding argument in the ellipsis argument list, and for some types, also controls the appearance of the output text. A subset of the possible type specifiers are given below. Note that type specifiers are case sensitive (e.g. 'g' is interpreted differently to 'G').

- c single character; supplied argument may be of type int, long or char
- d, i signed integer; supplied argument may be of type int, long or char; unsigned integral types are converted to signed
- f signed floating point value, output in simple decimal format (e.g. -3.142); supplied argument may be of type float or double; *integral types (int, long etc) must be explicitly cast to float or double*
- e, E signed floating point value, output in exponent format (e.g. -1.23E+45); the symbol 'E' in the output value may be in upper or lower case, depending on the case of the specifier; supplied argument may be of type float or double; *integral types must be explicitly cast to float or double*
- g, G signed floating point value, output in either simple decimal or exponent format, depending on the value of the supplied argument; the symbol 'E', if included in the output value, may be in upper or lower case, depending on the case of the specifier; supplied argument may be of type float or double; *integral types must be explicitly cast to float or double.*
- s character string; supplied argument must be a char* pointer to the string to output; *Important:* strings of type xtring (or other non-pointer representations) must be explicitly cast to char* (see example below)

Example:

The following code:

```
xtring name="Bert Higgins";
int age=24;
double height=1.8;

FILE* out=fopen("logfile.txt","w");

fprintf(out, "Name: %s\nAge:%4d\nHeight:%5.2f\n",
        (char*)name, age, height);
```

results in output of the following text to the file logfile.txt:

```
Name: Bert Higgins
Age: 24
Height: 1.80
```

Closing files

Streams are closed using the **fclose** function. You should make it a habit to close all files explicitly, even though streams are normally closed automatically when a program terminates.

Function synopsis:

```
int fclose(FILE* stream)
```

The function returns null (0) except in the event of an error (e.g. if the specified FILE* is not a valid handle to an open stream).

Other useful input/output functions

Function synopsis:

```
int feof(FILE* stream)
```

This function returns null (0) unless the file pointer on the specified stream has passed the last data byte in the file, i.e. end-of-file has been reached. Function **feof** is often useful when reading a file from start to finish, as in the following example.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <gutil.h>

double lon,lat,temp[12];

FILE* in=fopen("temp.txt","r");
FILE* out=fopen("temp.bin","wb");

while (!feof(in)) {
    if (benutil::readfor(in,"f7.3,f6.2,12f6.1",&lon,&lat,temp)) {
        fwrite(&lon,sizeof(int),1,out);
        fwrite(&lat,sizeof(int),1,out);
        fwrite(temp,sizeof(double),12,out);
    }
}

fclose(in);
fclose(out);
```

Function synopsis:

void rewind(FILE* *stream*)

Repositions the file pointer on the specified stream to the start of the file.

Example:

```
bool findcell(double lon,double lat,FILE* in,double temp[12]) {  
  
    // Finds the record corresponding to grid cell (lon,lat) on the  
    // specified stream and reads temperature data for this grid cell to  
    // array temp.  
    // Returns false if the record could not be found  
  
    double rlon,rlat;  
    bool rewound=false;  
  
    while (!feof(in)) {  
        if (benutil::readfor(in,"f7.3,f6.2,12f6.1",&rlon,&rlat,temp)) {  
            if (rlon==lon && rlat==lat) return true;  
        }  
        else if (!rewound) { // rewind if end-of-file reached  
            rewind(in);  
            rewound=true;  
        }  
    }  
    return false;  
}
```

Graphical output

More recent versions of LPJ-GUESS define a number of global functions for graphical output. These are functional only if LPJ-GUESS is built as a dynamic-link library (DLL) under Windows and run from the LPJ-GUESS windows shell. In other implementations the graphical functions may still be called but will not cause anything to happen.

Function synopsis:

void **plot**(xtring *window*, xtring *series*, double *x*, double *y*)

Opens a new graphics window called *window* if it is not already open; creates a new series (curve) called *series* if it does not already exist; adds a data point with the coordinate (*x*, *y*). Typically *x* will correspond to the simulation year (`date.year`) and *y* to the value of some state variable.

Example:

The following code, called every 10th simulation year, causes display of a chart with a curve corresponding to each of the soil organic matter pools in a certain patch:

```
if (!(date.year%10)) {
    plot("SoilC", "Fast", date.year, patch.soil.cpool_fast);
    plot("SoilC", "Slow", date.year, patch.soil.cpool_slow);
}
```

Function synopsis:

void **resetwindow**(xtring *window*)

Clears all series and data from window *window* but does not close the window.

Function synopsis:

void **clear_all_graphs**()

Clears all series and data from all open graphics windows, and closes the windows.

Custom library reference:**Character string manipulation with class xtring**

Most character strings in LPJ-GUESS are stored as objects of class xtring. The definition of class xtring and its member functions is included in the gutil library (gutil.lib; Unix version: gutil.a) which is a required to build LPJ-GUESS.

Construction and initialisation

The following forms of constructor function are supplied for constructing and initialising new xtring objects:

```
xtring()
xtring(char* str)
xtring(char ch)
xtring(int initsize)
```

If a char* or char argument is supplied, it provides the initial string text; otherwise an empty string ("") is assigned. If the integer argument *initsize* is given, the empty string is assigned to the object, but at least *initsize*+1 bytes of memory are set aside for the object's string buffer (this implies that a string up to *initsize* characters in length can subsequently be assigned to the object without reallocation of memory for the string buffer).

Declare new xstring objects using one of the following forms:

```
xtring s; // equivalent to xtring s=""
xtring s="initial text";
xtring s='c';
xtring s(INITSIZE);
```

You can also cast a string or character literal to an xtring the usual way:

```
(xtring)"Cast to a xtring"
(xtring)'c'
```

In general, xtring objects can be used in place of standard C char* strings without explicit casting, e.g.

```
char copy[100];
xtring original="text";
strcpy(copy,original);
```

However, xtring objects must be explicitly casted to char* when specified as arguments in calls to functions with an ellipsis argument, e.g.

```
xtring name="Ben";
printf("My name is: %s", (char*)name);
```

Casting to char* is useful also if you (unwisely?) choose to write directly to the internal string buffer of the xtring object:

```
xtring name(100);
char* pbuffer=(char*)name;
strcpy(pbuffer, "Ben");
```

Note, however, that some of the member functions of `xtring` can cause the size and memory position of the internal buffer to change. You must ensure that the buffer is at least `size+1` bytes (characters) in length if writing a string `size` bytes in length to it. The specified minimum size of the buffer is guaranteed if the `xtring` object is initialised using the `xtring(int)` constructor (see above) or following a call to member function `reserve` (see below).

Other public member functions

`unsigned long len()`

Returns the length of the current string in characters (not including a trailing null character); e.g.

```
xtring s="18 characters long";
int result=s.len();
```

`xtring upper()`

Returns a new `xtring` equivalent to the current one, but with lower-case alphabets 'a'-'z' converted to upper case; e.g.

```
xtring s="the quick brown fox";
xtring t=s.upper(); // t set to "THE QUICK BROWN FOX"
```

`xtring lower()`

Returns a new `xtring` equivalent to the current one, but with upper-case alphabets 'A'-'Z' converted to lower case; e.g.

```
xtring s="THE QUICK BROWN FOX";
xtring t=s.lower(); // t set to "the quick brown fox"
```

`xtring printable()`

Returns a new `xtring` equivalent to the current one, but with non-printable characters (ASCII code 0-31) removed.

`xtring left(unsigned long n)`

Returns a new `xtring` consisting of the leftmost `n` characters of the current `xtring`. An empty string ("") is returned if $n \leq 0$; if $n >$ length of the current `xtring`, an identical copy of the current `xtring` is returned; e.g.

```
xtring s="Ben Smith";
xtring t=s.left(3); // t set to "Ben"
```

`xtring mid(unsigned long s, unsigned long n)`
`xtring mid(unsigned long s)`

If both arguments are given, returns a new xtring consisting of up to n characters, starting at character number s (zero-based) of the current xtring. If $s < 0$, the new string starts at character 0 of the current xtring; if $s \geq$ length of the current xtring, an empty string is returned. If argument n is omitted, returns a new xtring consisting of the rightmost portion of the current xtring, starting at character number s . If $s < 0$ an identical copy of the current xtring is returned; e.g.

```
xtring s="Wolfgang Amadeus Mozart";
xtring t=s.mid(9); // t set to "Amadeus Mozart"
xtring q=s.mid(9,7); // q set to "Amadeus"
```

xtring **right**(unsigned long n)

Returns a new xtring consisting of the rightmost n characters of the current xtring. An empty string is returned if $n \leq 0$; if $n >$ length of the current xtring, an identical copy is returned; e.g.

```
xtring s="Ben Smith";
xtring t=s.right(5); // t set to "Smith"
```

long **find**(char* s)

long **find**(char c)

Returns the position (zero based) of the specified character string or character if it occurs as a substring of the current xtring. If a char* argument longer than one character is given, the returned value is the position of the first character of the substring, if it is found. If there are several occurrences, the position of the leftmost occurrence is returned. Returns -1 if the string or character is not found; e.g.

```
xtring s="ababcd";
int n=s.find('b'); // n set to 1
int m=s.find("bc"); // m set to 2
int q=s.find("de"); // q set to -1
```

long **findoneof**(char* s)

Returns the position in this xtring of the first (leftmost) occurrence any character forming part of the string pointed to by s . Returns -1 if there are no occurrences; e.g.

```
xtring s="ababcd";
int n=s.findoneof("edc"); // n set to 3
```

long **findnotoneof**(char* s)

Returns the position in this xtring of the first (leftmost) character *not* forming part of the string pointed to by s . Returns -1 if no such character is found; e.g.

```
xtring s="ababcd";
int n=s.findnotoneof("abc"); // n set to 4
```

double num()

Returns the numerical value of the current xtring, if it is a valid representation of a double precision floating point number in C++. Call function `isnum()` to test whether the returned value is meaningful; e.g.

```
xtring pi="3.142";
double v=pi.num(); // v set to 3.142
```

char isnum()

Returns 1 if the current xtring is a valid representation of a double precision floating point number in C++, 0 otherwise; e.g.

```
xtring good="3.142";
xtring bad="three point one four two";
double val;
if (good.isnum()) val=good.num();
else val=bad.num();
```

void printf(char* fmt,...)

A printf-style function for writing formatted data to this xtring object. Equivalent to function `sprintf` in the standard C stream input/output library (`stdio.h`). See any standard C/C++ reference manual for full documentation of printf-style output functions; e.g.

```
xtring out;
char* name[]="pi";
double dval=3.142;
int ndec=3;
out.printf("%s has the value %g to %d decimal places",name,dval,ndec);
// out set to "pi has the value 3.142 to 3 decimal places"
```

void reserve(unsigned long n)

Expands or contracts the memory allocation to the current xtring object to accomodate a string at least *n* characters in length (not including the trailing null byte). The currently stored string may be copied to a new location in memory but is not deleted. This function may be useful if you intend to write directly to the internal string buffer, whose address is returned by casting the xtring object to `char*`; e.g.

```
xtring s;
s.reserve(100);
strcpy(s,"A string up to 100 characters long");
```

In general, however, there should be no reason to write directly to the internal string buffer of an xtring object; use the assignment (=) operator (see below) to assign a new value to the object.

Overloaded operators

The following operators are defined for xtring objects:

Assignment: =, +=

Concatenation: +, +=

Comparison: ==, !=, <, >, <=, >=

Array subscript: []

Operator functionality is described here mainly by code examples. The examples below assume the following data types for variables:

```
xtring x1, x2, x3;
char* s;
char c;
unsigned long n;
```

Assignment:

```
x1="Ben";
x1+=" Smith"; // x1 set to "Ben Smith"
```

Concatenation (appends a char* string, xtring or character to the end of an xtring string):

```
x1="Wolfgang ";
x2=" Mozart";
c='A';
x3=x1+c;
x3+=x2+" (composer)";
// x3 set to "Wolfgang A Mozart (composer)"
```

Concatenation of, for example, two char* strings, or a xtring to the end of a char* string, is possible by casting one of the operands to an xtring:

```
x1=(xtring) "Wolfgang "+"Mozart";
x1=(xtring) "Wolfgang"+x2;
```

Comparison (== and != compare string identity; <, >, <=, >= compare "alphabetic" rank):

```
x1==x2 && x2!=x3 || x1<s || x1>c || x2<=x3 || x3>=x1
```

Note that the left hand operand must be a xtring (or casted to xtring)

Array subscript ($x[n]$ retrieves a reference to the n th character [zero-based] within xtring x):

```
c=x1[n];
x1[n]=c;
```

The size of the internal string buffer is expanded if necessary to ensure that the specified subscript is valid (points to a character position within the internal string buffer). However, the string itself is not expanded (i.e. the position of the trailing null byte, signifying the end of the string, is not changed).

Appendix: Keywords recognised in the instruction script

The following is a list of keywords and their meanings recognised in instruction scripts (ins files) by the CRU input/output module of LPJ-GUESS. In nearly all cases, the keyword corresponds to the name of the variable in the model code to which the associated value is assigned. Custom "param" settings are not included in this list.

General (global) settings

May appear anywhere in the ins file, except within "group", "pft" or "param" blocks.

title	Title for run
nyear_spinup	Number of simulation years to spinup for
vegmode	Vegetation mode ("INDIVIDUAL", "COHORT", "POPULATION")
ifbgestab	Whether background establishment enabled (0,1)
ifsme	Whether spatial mass effect enabled for establishment (0,1)
ifstochmort	Whether mortality stochastic (0,1)
ifstochestab	Whether establishment stochastic (0,1)
estinterval	Interval for establishment of new cohorts (years)
distinterval	Generic patch-destroying disturbance interval (years)
iffire	Whether fire enabled (0,1)
ifdisturb	Whether generic patch-destroying disturbance enabled (0,1)
ifcalcsla	Whether SLA calculated from leaf longevity
ifcalctcon	Whether leaf C:N ratio min calculated from leaf longevity
ifcdebt	Whether to allow C storage
npatch	Number of patches simulated (for natural landcover)
patcharea	Patch area (m ²)
wateruptake	Water uptake mode ("WCONT", "ROOTDIST", "SMART", "SPECIESSPECIFIC")
ifcentury	Whether to use CENTURY SOM dynamics (mandatory for N cycling)
ifnlim	Whether plant growth limited by leaf N
freenyears	Number of years to spin up without N limitation (needed to build up a N pool)
nfix_a	First term in N fixation eqn
nfix_b	Second term in N fixation eqn
nrelocfrac	Fraction of N retranslocated prior to leaf and root shedding
outputdirectory	Directory for the output files
file_cmass	C biomass output file
file_anpp	Annual NPP output file
file_aiso	Annual isoprene emissions output file
file_amon	Annual monoterpene emissions output file
file_lai	LAI output file
file_cflux	C fluxes output file
file_dens	Tree density output file
file_cpool	Soil C output file
file_runoff	Runoff output file
file_firert	Fire return time output file
file_cton_leaf	Leaf C:N ratio output file

file_cton_veg	Vegetation C:N ratio output file
file_nsources	Annual N sources output file
file_npool	Soil N output file
file_nuptake	Annual N uptake output file
file_nflux	N fluxes output file
file_ngases	Annual N gases output file
file_vmaxnlim	Annual average N limitation on Vmax output file
file_mnpp	Monthly NPP output file
file_mlai	Monthly LAI output file
file_mgpp	Monthly GPP-LeafResp output file
file_mra	Monthly autotrophic respiration output file
file_maet	Monthly AET output file
file_mpet	Monthly PET output file
file_mevap	Monthly Evap output file
file_mrunoff	Monthly runoff output file
file_mintercep	Monthly intercep output file
file_mrh	Monthly heterotrphic respiration output file
file_mnee	Monthly NEE output file
file_mwcont_upper	Monthly wcont_upper output file
file_mwcont_lower	Monthly wcont_lower output file
file_miso	Monthly isoprene emissions output file
file_mmon	Monthly monoterpene emissions output file
ifsmoothgreffmort	Whether to vary mort_greff smoothly with growth efficiency (0,1)
ifdroughtlimitedestab	Whether establishment drought limited (0,1)
ifrainonwetdaysonly	Whether it rains on wet days only (1), or a little every day (0);
ifbvoc	Whether or not to include BVOC calculations (0,1)
searchradius	If specified, CRU data will be searched for in a circle
pft	Header for block defining PFT
run_landcover	Whether other landcovers than natural vegetation are simulated.
run_urban	Whether urban land is to be simulated
run_crop	Whether crop-land is to be simulated
run_pasture	Whether pasture is to be simulated
run_forest	Whether managed forest is to be simulated
run_natural	Whether natural vegetation is to be simulated
run_peatland	Whether peatland is to be simulated
lfrac_fixed	Whether landcover fractions are read from ins-file.
equal_landcover_area	Whether gridcell is divided into equal active landcover fractions.
lc_fixed_urban	Fixed urban landcover fraction
lc_fixed_cropland	Fixed crop-land landcover fraction
lc_fixed_pasture	Fixed pasture landcover fraction
lc_fixed_forest	Fixed managed forest landcover fraction
lc_fixed_natural	Fixed natural landcover fraction
lc_fixed_peatland	Fixed peatland landcover fraction
state_path	State files directory (for restarting from, or saving state files)
restart	Whether to restart from state files
save_state	Whether to save new state files
state_year	Save/restart year. Unspecified means just after spinup

PFT settings

Must appear within a named "pft" block, or "group" block. In the latter case, the group name should appear as a setting in the associated pft block(s).

include	Include PFT in analysis
lifeform	Lifeform ("TREE" or "GRASS")
phenology	Phenology ("EVERGREEN", "SUMMERGREEN", "RAINGREEN" or "ANY")
landcover	Landcover ("URBAN", "CROPLAND", "PASTURE", "FOREST", "NATURAL" or "PEATLAND").
leafphysiognomy	Leaf physiognomy ("broadleaf" or "needleleaf")
phengdd5ramp	GDD on 5 deg C base to attain full leaf cover
wscal_min	Water stress threshold for leaf abscission (raingreen PFTs)
pathway	Biochemical pathway ("C3" or "C4")
pstemp_min	Approximate low temp limit for photosynthesis (deg C)
pstemp_low	Approx lower range of temp optimum for photosynthesis (deg C)
pstemp_high	Approx higher range of temp optimum for photosynthesis (deg C)
pstemp_max	Maximum temperature limit for photosynthesis (deg C)
lambda_max	Non-water-stressed ratio of intercellular to ambient CO2 pp
rootdist	Fraction of roots in each soil layer (first value=upper layer)
gmin	Canopy conductance not assoc with photosynthesis (mm/s)
emax	Maximum evapotranspiration rate (mm/day)
respcoeff	Respiration coefficient (0-1)
cton_root	Fine root C:N mass ratio (used in respiration)
cton_sap	Sapwood C:N mass ratio (used in respiration)
reprfrac	Fraction of NPP allocated to reproduction
turnover_leaf	Leaf turnover (fraction/year)
turnover_root	Fine root turnover (fraction/year)
turnover_sap	Sapwood turnover (fraction/year)
wooddens	Sapwood and heartwood density (kgC/m3)
crownarea_max	Maximum tree crown area (m2)
k_allom1	Constant in allometry equations
k_allom2	Constant in allometry equations
k_allom3	Constant in allometry equations
k_rp	Constant in allometry equations
k_latosa	Tree leaf to sapwood xs area ratio
sla	Specific leaf area (m2/kgC)
ltor_max	Non-water-stressed leaf: fine root mass ratio
litterme	Litter moisture flammability threshold (fraction of AWC)
fireresist	Fire resistance (0-1)
tmin_surv	Min 20-year coldest month mean temp for survival (deg C)
tmin_est	Min 20-year coldest month mean temp for establishment (deg C)
tcmx_est	Max 20-year coldest month mean temp for establishment (deg C)
twmin_est	Min warmest month mean temp for establishment (deg C)

twminusc	Stupid larch parameter
gdd5min_est	Min GDD on 5 deg C base for establishment
k_chilla	Constant in equation for budburst chilling time requirement
k_chillb	Coefficient in equation for budburst chilling time requirement
k_chillk	Exponent in equation for budburst chilling time requirement
parff_min	Min forest floor PAR for grass growth/tree estab (J/m ² /day)
alphar	Shape parameter for recruitment-juv growth rate relationship
est_max	Max sapling establishment rate (indiv/m ² /year)
kest_repr	Constant in equation for tree estab rate
kest_bg	Constant in equation for tree estab rate
kest_pres	Constant in equation for tree estab rate
longevity	Expected longevity under lifetime non-stressed conditions (yr)
greff_min	Threshold for growth suppression mortality (kgC/m ² leaf/yr)
leaflong	Leaf longevity (years)
intc	Interception coefficient
drought_tolerance	Drought tolerance level (0 = very -> 1 = not at all) (unitless)
ga	Aerodynamic conductance of leaf (m/s)
eps_iso	Isoprene emission capacity (ugC/g/h)
seas_iso	Whether or not isoprene emissions show a seasonality (0,1)
eps_mon	Monoterpene emission capacity (ugC/g/h)
storfrac_mon	Fraction of monoterpene production that goes into storage pool (unitless)
nuptoroot	Maximum nitrogen uptake per fine root
km_volume	Half saturation concentration for N uptake [kgN l ⁻¹]
cton_leaf_min	Minimum leaf C:N ratio allowed (maximum N concentration)
fnstorage	Fraction of sapwood (root for herbaceous pfts) that can be used as a nitrogen longterm storage scalar

References

- Arneth, A., Niinemets, Ü., Pressley, S., Bäck, J., Hari, P., Karl, T., Noe, S., Prentice, I.C., Serca, D., Hickler, T., Wolf, A. & Smith, B. 2007. Process-based estimates of terrestrial ecosystem isoprene emissions: incorporating the effects of a direct CO₂-isoprene interaction. *Atmospheric Chemistry and Physics* 7: 31-53.
- Bachelet, D., Neilson, R.P., Hickler, T., Drapek, R.J., Lenihan, J.M., Sykes, M.T., Smith, B., Sitch, S. & Thonicke, K. 2003. Simulating past and future dynamics of natural ecosystems in the United States. *Global Biogeochemical Cycles* 17: 1045-1065.
- Badeck, F.-W., Lischke, H., Bugmann, H., Hickler, T., Hönniger, K., Lasch, P., Lexer, M.J., Mouillot, F., Schaber, J. & Smith, B. 2001. Tree species composition in European pristine forests. Comparison of stand data to model predictions. *Climatic Change* 51: 307-347.
- Cramer, W., Bondeau, A., Woodward, F.I., Prentice, I.C., Betts, R.A., Brovkin, V., Cox, P.M., Fisher, V., Foley, J.A., Friend, A.D., Kucharik, C., Lomas, M.R., Ramankutty, N., Sitch, S., Smith, B., White, A. & Young-Molling, C. (2001) Global response of terrestrial ecosystem structure and function to CO₂ and climate change: results from six dynamic global vegetation models. *Global Change Biology* 7: 357-373.
- Cramer, W., Bondeau, A., Schaphoff, S., Lucht, W., Smith, B. & Sitch, S. 2004. Tropical forests and the global carbon cycle: Impacts of atmospheric CO₂, climate change and rate of deforestation. *Philosophical Transactions of the Royal Society of London, Series B*. 359: 331-343.
- Foley, J.A. 1995. An equilibrium model of the terrestrial carbon budget. *Tellus* 47B: 310-319.
- Fulton, M.R. 1991. Adult recruitment rate as a function of juvenile growth in size-structured plant populations. *Oikos* 61: 102-105.
- Gerten, D., Schaphoff, S., Haberlandt, U., Lucht, W. & Sitch, S. 2004. Terrestrial vegetation and water balance – hydrological evaluation of a dynamic global vegetation model. *Journal of Hydrology* 286: 249-270.
- Haxeltine A. & Prentice I.C. 1996. BIOME3: an equilibrium terrestrial biosphere model based on ecophysiological constraints, resource availability, and competition among plant functional types. *Global Biogeochemical Cycles* 10: 693-709.
- Hickler, T., Smith, B., Sykes, M.T., Davis, M.B., Sugita, S. & Walker, K. 2004. Using a generalized vegetation model to simulate vegetation dynamics in the western Great Lakes region, USA, under alternative disturbance regimes. *Ecology* 85: 519-530.
- Lloyd, J. & Taylor J.A. 1994. On the temperature dependence of soil respiration. *Functional Ecology* 8: 315-323.
- Lucht, W., Prentice, I.C., Myneni, R.B., Sitch, S., Friedlingstein, P., Cramer, W., Bousquet, P., Buermann, W. & Smith, B. 2002. Climatic control of the high-latitude vegetation greening trend and Pinatubo effect. *Science* 296: 1687-1689.
- Prentice, I.C., Sykes, M.T. & Cramer W. 1993. A simulation model for the transient effects of climate change on forest landscapes. *Ecological Modelling* 65: 51-70.
- Reich, P.B., Walters M.B. & Ellsworth D.S. 1997. From tropics to tundra: global convergence in plant functioning. *Proceedings of the National Academy of Sciences USA* 94: 13730-13734.
- Sitch, S., Prentice I.C., Smith, B. & Other LPJ Consortium Members, 2000. LPJ – a coupled model of vegetation dynamics and the terrestrial carbon cycle. In: Sitch, S. *The Role of Vegetation Dynamics in the Control of Atmospheric CO₂ Content*, Ph.D. Thesis, Lund University, Lund, Sweden.
- Sitch, S., Smith, B., Prentice, I.C., Arneth, A., Bondeau, A., Cramer, W., Kaplan, J., Levis, S., Lucht, W., Sykes, M., Thonicke, K. & Venevsky, S. 2003. Evaluation of ecosystem dynamics, plant geography and terrestrial carbon cycling in the LPJ Dynamic Global Vegetation Model. *Global Change Biology* 9: 161-185.
- Smith, B., Prentice, I.C. & Sykes, M. 2001. Representation of vegetation dynamics in the modelling of terrestrial ecosystems: comparing two contrasting approaches within European climate space. *Global Ecology and Biogeography* 10: 621-637.
- Sykes, M.T., Prentice I.C. & Cramer W. 1996. A bioclimatic model for the potential distributions of north European tree species under present and future climates. *Journal of Biogeography* 23: 209-233.
- Sykes, M.T., Prentice, I.C., Smith, B., Cramer, W. & Venevsky, S. 2001. An Introduction to the European Terrestrial Ecosystem Modelling Activity. *Global Ecology and Biogeography* 10: 581-594.
- Thonicke, K., Venevsky, S., Sitch, S. & Cramer, W. 2001. The role of fire disturbance for global vegetation dynamics: coupling fire into a Dynamic Global Vegetation Model. *Global Ecology and Biogeography* 10: 661-677.

Websites

www.pik-potsdam.de/lpj

www.nateko.lu.se/lpj-guess