**Cryogrid**

**Table of contents**

1. How to Set Up a New Model Run
2. Structure of the Model and Terminology
3. Style Guide
4. How to Write a New Forcing Class
5. How to Write a New GROUND class
6. How to Write a New InterAction (IA) class
7. How to write a new LATERAL class

**Quick Start - how to set up a new 1D model run**

1. Create a new folder *yourRunningNumber* in the results folder
2. Create (or copy) the files *CONSTANT\_excel.xlsx* and *yourRunningNumber.xlsx* in that folder
3. Check if all *constants* you need are set up in *CONSTANT\_excel.xlsx*
4. Change the runningNumber in *run\_CG\_1D.m* to *yourRunningNumber*
5. Set up a parameter file in *yourRunningNumber.xlsx*
6. Set up all necessary *parameters* and *state variables* for the classes that you are using in the stratigraphy in *yourRunningNumber.xlsx*
7. Select the desired LATERAL classes and set up all necessary parameters for them in *yourRunningNumber.xlsx*
8. Select the desired OUT (i.e. output) class which determines the output formats
9. Select the desired FORCING class which determines the input data
10. Run main program *run\_CG\_1D\_excel.m*

For running from parameter files in YAML format, the procedure is similar

**Terminology**

* *parameter file*: spreadsheet (in Excel and YAML format at this point) used to set up a CryoGrid run
* *class*: Matlab class (<https://se.mathworks.com/help/matlab/object-oriented-programming.html>), realizes object-oriented programming in Matlab
* *CryoGrid GROUND class*: Matlab class with mandatory functions and properties which contains the defining equations for the time evolution of the ground, snow, lake etc. Note that many of the classes in the code are not CryoGrid *GROUND* classes, but do something else! A *module* is a realization of the class.
* *CryoGrid INTERACTION (IA) class*: Matlab class with mandatory functions and properties which contains the defining equations for interactions and fluxes between pairs of GROUND classes. Two GROUND classes compatible with each other, if an IA class is available for them.
* *CryoGrid stratigraphy*: stratigraphy of vertically connected CryoGrid GROUND classes with IA classes between each pair of GROUND classes.
* *CryoGrid tile:* a model realization defined by its stratigraphy, model forcing, etc.
* *CryoGrid LATERAL class*: Matlab class with mandatory functions and properties which contains the defining equations for lateral interactions and fluxes, either for a single tile (1D) or multi-tile (3D) runs. LATERAL classes act on the entire Cryogrid stratigraphy, i.e. they exchange information with all GROUND classes in the stratigraphy.

**Folder structure**

* *run\_CG\_1D.m* : main file to be executed for single tile (1D) model runs
* **modules**: contains the model code
* **modules/TIER0**: base level: contains the base code for CryoGrid ground and interaction classes. TIER0 does not contain functional CryoGrid classes.
* **modules/TIER1**: library level: inherits from TIER0 base classes, contains classes comprising all functions related to a certain physical process. TIER1 does not contain functional CryoGrid classes.
* **modules/TIER2:** first GROUND class level: inherits from TIER1 library classes**,** contains fully functional CryoGrid classes
* **modules/TIER3:** second GROUND class level: inherits from TIER2 GROUND classses**,** contains fully functional CryoGrid classes. TIER3 in particular contains the GROUND classes that can interact with a SNOW class
* **modulesTIER**XX**/INTERACTION:** INTERACTON IA classes defining interactions and fluxes between pairs of GROUND classes, same TIER structure as for GROUND classes
* **modules/LATERAL:** contains classes for lateral interactions and fluxes in subfolders for single tile (1D) and multi-tile (3D) runs
* **modules/FORCING**: contains FORCING classes, which read forcing data and provide it to the GROUND classes
* **modules/BUILDERS:** contains BUILDER classeswhich assemble the initial state including the stratigraphy for the CryoGrid model run. BUILDER classes use the information from the different PROVIDER classes as input (TIN, JS, please correct).
* **modules/IO:** all functionality related to model input and output
* **modules/IO/PARAMETER\_PROVIDER:** provider classes reading the information from the parameter file. (TIN, JS, please correct).
* **modules/IO/CONSTANT\_PROVIDER:** provider classes reading the information from the CONSTANT file containing physical constant. (TIN, JS, please correct).
* **modules/IO/FORCING\_PROVIDER:** provider classes reading the information from the parameter file. (TIN, JS, please correct).
* **modules/IO/GRID:** GRID classes offer different possibilities to define the model grid
* **modules/IO/STRATIGRAPHY**: STRATIGRAPHY classes provide different options to assign state variables to the model grid.
* **modules/IO/OUT**:OUT classes provide different options to store model output
* **modules/IO/INITIALIZE\_PARAMETER\_FILE:** (TIN, JS, should we move this to BUILDERS?).
* **results**: contains the parameter file and the output files in a folder corresponding to the variable *run\_number* in *run\_CG\_1D.m*
* **forcing**: contains the forcing data as *.mat*-files. If you use the FORCING class, the files must have the same structure as in the sample file provided.
* **analyze\_display**: contains service functions to display model output, such as *read\_display\_out.m*, which displays the module *out* produced by the OUT class *OUT\_all.m* (all CryoGrid classes stored in *out* must be initialized prior to calling *read\_display\_out.m*).

**TIER levels and inheritance**

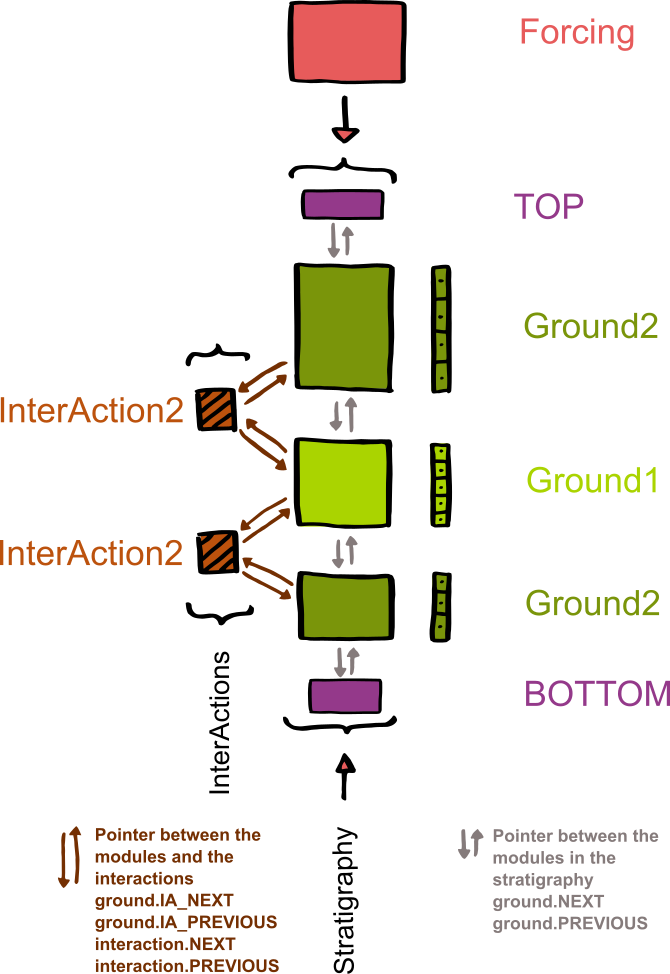
The TIER levels 0 to 3 provide a structure for creating new GROUND classes, while at the same time reducing redundancy of the code. The following rules apply:

1. The TIER0 BASE classes provides the mandatory variables and mandatory functions; in general, they are empty arrays/ empty functions. All BASE classes must inherit *matlab.mixin.Copyable*, so that it is possible to make identical copies of all classes using *copy()*.
2. TIER1 library classes must inherit from the respective BASE class. It is possible to define additional variables and functions. When defining a new variable, it *must* have a unique name that is different from the variables in all other TIER1 classes. In TIER1, classes comprise all functions related to specific processes, such as heat conduction, water fluxes, etc. There is clearly some redundancy within the function in some of the classes, but the idea is to have as much as possible of the code within each function to make it easy to understand.
3. TIER2 GROUND classes contain all mandatory functions that are called in the main file, plus optional functions in addition. TIER2 classes inherit from *several* (!!) TIER1 library classes, depending on processes represented. It is possible to define from additional variables. In TIER2, all classes are fully functional CryoGrid GROUND classes (which include classes representing snow, lakes, etc.)
4. TIER3 GROUND classes inherit from generally one TIER2 class, but add more functionality. At this stage, coupling to snow cover classes is added in TIER3, but other applications might arise in the future.
5. INERACTION classes follow a similar structure with TIER0 to TIER2

**Basic structure of the subsurface stratigraphy**

Stacking of different CryoGrid GROUND classes to form a stratigraphy is organised with pointers to the NEXT and PREVIOUS CryoGrid classes, and dedicated TOP and BOTTOM classes as confining classes. In a similar way, the interaction classes (e.g. for exchange of heat, salt, water) are addressed by pointers from the two GROUND classes.

In detail, the stacked GROUND (i.e. ground, snow, lake, etc.) classes (denoted *ground\_class)* are linked with pointers *ground\_class.NEXT* and ground\_class.*PREVIOUS*. The pointers *ground\_class.IA\_NEXT* and *ground\_class.IA\_PREVIOUS* point to the *interaction\_class* while *interaction.NEXT* and *interaction.PREVIOUS* point to *ground\_class*. In this way, the INTERACTION class can directly access variables in both involved GROUND classes.

[](https://github.com/CryoGrid/CryoGrid/blob/GitHub_CryoGrid/NewOOP/develop/THIN/documentation/Overview_Modules_noSnow.png)

A GROUND class can delete itself, move itself to a new position, and initialize and insert a new module (from the same or different classes). This is for example used for snow buildup, or to switch between LAKE classes representing frozen and unfrozen lakes. For this purpose, the pointers of the different GROUND classes and INTERACTION classes involved must be repositioned in the correct order. A class is deleted (by the automatic garbage collection in Matlab) when all pointers from still existing classes/variables are deleted.

**Compatibility of GROUND classes**

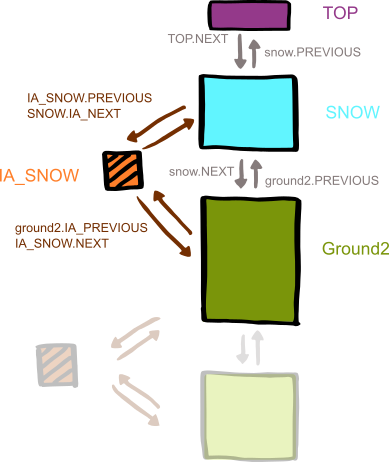
All GROUND classes can in principle handle whichever state variable or boundary condition. However, the state variables boundary conditions need to be compatible with the used interaction class(es) and the used forcing. For example, if you have temperature as a state variable and the interaction between two modules is handled as a heat flow, then the INTERACTION class must compute heat flows between the two modules.

The function *get\_IA\_class.m*  in modules/TIER2/INTERACTION/ is used to determine the correct INTERACTION class for each pair of GROUND classes and create and initialize it. This is accomplished by a compatibility matrix in *get\_IA\_class.m.* If you write a new GROUND or INTERACTION class, please insert your use cases in this function.

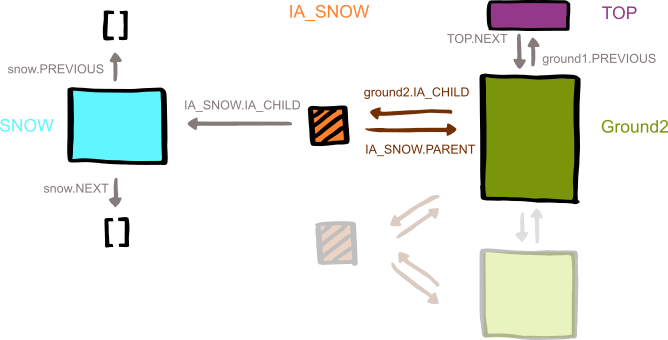
Note that the compatibility in general depends on the order of the two classes. For example, a snow class is only compatible with GROUND classes, when it is on top, i.e. appears above the GROUND class in the stratigraphy. The same is true for hydrologically active GROUND classes, which are only compatible with a non-active ground class when they appear above it in the stratigraphy. Most of the compatibility limitations are determined by common sense, or the applications of the model. If desired, it is possible to make two classes compatible by creating a dedicated INETRACTION class and updating *get\_IA\_class.m.*

**Seasonal build-up and disappearance of snow cover**

The snow cover is simulated by dedicated TIER2 GROUND classes *SNOW\_XX.m*. The basic principle is that the SNOW class is added between the TOP class and the uppermost GROUND class, when a snow cover is present. Interactions between the SNOW and GROUND class are again realized by dedicated INTERACTION classes, which facilitate the exchange of energy, water, etc., depending on the capabilities of the involved classes.

[](https://github.com/CryoGrid/CryoGrid/blob/GitHub_CryoGrid/NewOOP/develop/THIN/documentation/Overview_Modules_withSnow.png)

A significant problem for the numerical scheme is to handle very shallow snow covers, as this results in a small gird cell size and thus very small timesteps. In CryoGrid, the SNOW classes are attached and detached in two stages, after the first snowfall first as a CHILD of the uppermost GROUND class, then becoming a full class within the stratigraphy when enough snow has accumulated. In the CHILD phase, both the physics and the interactions with the GROUND class are handled by special CHILD functions, both in the SNOW class and in the INTERACTION class coupling the SNOW to the GROUND class. When the snow depth is large enough, so that it can be handled without numerical problems, the SNOW class and the associated INTERACTION class are simply rearranged, so that it is part of the regular stratigraphy. The procedure is repeated in the opposite direction during snowmelt.

[](https://github.com/CryoGrid/CryoGrid/blob/GitHub_CryoGrid/NewOOP/develop/THIN/documentation/Overview_Modules_buildingSnow.png)

**Style guide**

* **Most important (THIS IS A MUST!): Use clear and understandable variable names and not abbreviations**
* Physical properties should be named by their SI symbols (if it exists and makes sense).
* In functions, used variables from containers should be saved by their name, i.e. *theta = ground.CONST.theta*
* Variables and functions should be in lower case. For variable names consisting of several words, camel case or underscores can be used.
* If you use equations and constants, cite their source in comments.
* The CryoGrid code contains many cases in which the style guide was not followed, partly due to implementation of legacy code, partly due to negligence. To ensure readability of the code and error messages, rule No.1 is by far the most important and must be followed!!

**Documentation**

Every author should indicate her/his name and the date in the header of each class. If major changes or updates are done, the author(s) of the changes should again include name(s) and date, but not remove the previous entries in the header. In the code, comments should be inserted. In addition, each CryoGrid class should be described in this manual. Every author is responsible for the documentation of new CryoGrid class.

**GROUND classes**

Mandatory Functions and Variables

**INTERACTION classes**

Mandatory Functions and Variables

**LATERAL classes**

**Mandatory Functions and Variables**

*ground.CONST*

Need to be allocated in *provide\_CONST.m* for every class, or in the inheriting superclass(es). Need to be specified in *CONSTANT\_excel.xlsx*

year\_sec = 365\*24\*3600; % s / yr

Latent heat L\_f = 334\*1000\*1000; % J / m^3

Heat capacity c\_w = 4.2\*1000\*1000; % J / (K m^3)

c\_i = 1.9\*1000\*1000; % J / (K m^3)

c\_o = 2.5\*1000\*1000; % J / (K m^3)

c\_m = 2\*1000\*1000; % J / (K m^3)

Thermal conductivity

air k\_a = 0.025; % W / m K [Hillel(1982)]

water k\_w = 0.57; % W / m K [Hillel(1982)]

ice k\_i = 2.2; % W / m K [Hillel(1982)]

organic k\_o = 0.25; % W / m K [Hillel(1982)]

mineral k\_m = 3; % W / m K

quartz k\_q = 7; % W / m K

Stefan-Boltzmann const.

sigma = 5.67e-8;

Universal gas constant

R = 8.314459; % J / K mol

Tortuousity tau = 1.5; % 1.5 standard

Freezing temperature of water

Tmelt\_free\_water = 273.15; % K

*ground.PARA*

Need to be allocated in *provide\_PARA.m* for every class, or in the inheriting superclass(es). Need to be specified in *yourRunningNumber.xlsx* for every class within the block for that class.

Default heat flux at lower boundary

heatFlux\_lb = 0.05; % W / m^2

Albedo albedo = 0.15; %

Emissivity epsilon = 0.99; %

rs = 100; %

Measurement height

z =2; % m

Roughness length

z0 = 1e-3; % m

Maximum time step

dt\_max = 3600 ; % s

Maximum energy change per time step

dE\_max = 0.5e5; % J / m^3

Maximum temperature change per time step

dT\_max = 0.5e5; % K

Maximum salt concentration change per time step

dsaltConc\_max %

Snow water equivalent per cell

swe\_per\_cell = 0.01; % m must be the same as in SNOW class, should be taken from input file

Sedimentation Rate

sedRate % m / yr

Soil Type

soilType = 1 .\* ones(size(grid\_cell\_size)); %0 - sand, 1 - silt

*ground.STATVAR* (depends on your choice of state variables)

Need to be allocated in *provide\_STATVAR.m* for every class, or in the inheriting superclass(es). Need to be specified in *yourRunningNumber.xlsx* for the stratigraphy and in *finalize\_STATVAR.m* for every class.

upperPos = grid(1);

lowerPos = grid(end);

layerThick % the distances between layer boundaries in m

% Should be sufficient to save only upperPos, lowerPos and layerThick

% However, if you want to save other info, please use the following

midptDepth % the depths of module layer midpoints in m

midptThick % the distances between layer midpoints in m

layerDepth % the depths of the module layer boundaries in m

Initial fractional volumes

water = 0.4 .\* ones(size(grid\_cell\_size)); %fraction of volume "m^3/m^3"

ice = 0 .\* ones(size(grid\_cell\_size)); %fraction of volume

organic = 0 .\* ones(size(grid\_cell\_size)); %fraction of volume

mineral = 1.1 \* 0.6 .\* ones(size(grid\_cell\_size)); %fraction of volume

Salt diffusivity

saltDiff % m^2 / s

Bulk thermal conductivity

thermCond % W / m K

Effective heat capacity

c\_eff % J / m^3

Liquid water content

liqWater % fraction of volume

Freezing temperature of solution

Tmelt % K

% Of the following choose your state variables

Initial temperature

T % °C

Initial salinity - salt concentration in liquid water

saltConc = 895 .\* ones(size(grid\_cell\_size)); % mol / l [Dimitrenko 2011]

Initial energy

E % J

Initial total ions - salt concentration per layer volume

salt % mol

*ground.TEMP*

To be calculated in the appropriate functions (*get\_boundary\_condition\_*\*\*, *get\_derivatives\_prognostic*, interactions, etc.) - depends on choice of state variables - needed boundary conditions depend also on the chosen interaction

% Boundary conditions for the chosen state variable(s) -

Temperature at upper boundary (for updating of thermal properties)

T\_ub % °C

Heat flux at upper boundary

heatFlux\_ub % W / m^2

Heat flux at lower boundary

heatFlux\_lb % W / m^2

Salt flux at upper boundary

saltFlux\_ub %

Salt flux at lower boundary

saltFlux\_lb %

Temperature divergence

divT %

Salt concentration divergence

divsaltConc %

*forcing.PARA*

Need to be specified in *yourRunningNumber.xlsx* in the forcing block. Other parameter of the forcing depends on the chosen forcing class.

Starttime of forcing data

startForcing

Time step of forcing data

dtForcing

Endtime of forcing data

endForcing

% Other depending on your forcing data

*forcing.DATA*

to be calculated in *generateForcing*. All variable are vectors with the same length

Vector of discrete time steps

timeForcing = [startForcing:dtForcing:endForcing]';

Vector of forcing temperature

TForcing

Vector of forcing salt concentration

saltConcForcing

*forcing.TEMP*

to be calculated in *interpolateForcing* - depending on the choice of state variables. All variables are single values - interpolated for each time step

Surface temperature at this time point

TForcing

Surface salt concentration at this time point

saltConcForcing

*OUT* und *RUN\_INFO*

Save interval dtSave

**FORCING classes**

A new forcing module loads or generates forcing data. The state variables that are provided by the forcing module need to match the ones that are needed by the used ground modules.

%> here goes the documentation of the focing module

classdef FORCING

properties

DATA %all data

TEMP %at each timestep

PARA

STATUS %forcing data suitable for the modules that are to be run -> can be used

end

methods %mandatory functions

%> Descitption of the functions, input and output parameters

function forcing = initalize\_from\_file(forcing, section)

%This function initializes the parameter from the section of the excel file to forcing.PARA

end

%> Descitption of the functions, input and output parameters

function forcing = load\_forcing\_from\_mat(forcing)

%This function loads/generates the forcing and stores it in forcing.DATA

end

%> Descitption of the functions, input and output parameters

function forcing = interpolate\_forcing(t, forcing)

%This function will be called by ground modules and should return all state variables, that are needed

end

end

end

**INTERACTION (IA) classes**

The interaction classes handle the boundary conditions for all modules in the stratigraphy that share an *inner boundary* with another module. The interaction has pointers to the two interacting modules and can therefore change both modules directly. The interaction needs to match the boundary conditions that are used in the modules. The interaction classes get selected in **/modules/INTERACTION/** *get\_IA\_class.m* based on the combination of modules. If you write a new class or interaction, please insert your use cases in this function.

%> Here comes the documentation of your interaction class

classdef IA\_HEAT < matlab.mixin.Copyable

properties

PREVIOUS

NEXT

end

methods

%> Descitption of the functions, input and output parameters

function get\_boundary\_condition\_m(ia\_heat)

stratigraphy1 = ia\_heat.PREVIOUS;

stratigraphy2 = ia\_heat.NEXT;

%put interaction here

%it should act as the boundary conditions for both involved section,

%i.e. get\_boundary\_condition\_l for stratigraphy1 and as get\_boundary\_condition\_u for stratigraphy2

end

end

end

**GROUND classes**

The definition of a new module should look like follows. The functions are mandatory for the time loop to work. The "initialize" function may take additional input parameters, as you only use it before entering the time loop, the other functions may not.

%> Here comes the documentation of your new module

classdef GROUND\_yourName

properties

CONST %constants

PARA %external service parameters, all other

STATVAR %state variables - choose wisely

TEMP %derivatives in prognostic

PREVIOUS %pointer to previous module

NEXT %pointer to next module

IA\_PREVIOUS %pointer to interaction with previous module

IA\_NEXT %pointer to interaction with next module

end

methods %mandatory functions for each class

%> Descitption of the functions, input and output parameters

function ground = provide\_variables(ground)

%initializes the variables in ground.CONST, ground.PARA and ground.STATVAR as empty arrays.

%You may also use this function from the superclass

end

function variable = initialize\_from\_file(ground, variable, section)

%fills variables in ground.CONST and ground.PARA with info from the appropriate section in the excel file

end

function ground = assign\_global\_variables(ground, forcing)

%assign variables from forcing to module

end

function ground = initialize\_STATVAR\_from\_file(ground, grid, forcing, depths)

%initialize state variables in ground.STATVAR with info from the appropriate section in the excel file

end

%> Descitption of the functions, input and output parameters

function ground = get\_boundary\_condition\_u(ground, forcing)

%put upper boundary here

%this function needs to be compatible with the used interaction class, i.e. if you use heatflux here,

%you also need to use it in the interaction class

end

%> Descitption of the functions, input and output parameters

function ground = get\_boundary\_condition\_l(ground)

%put lower boundary here

end

function ground = get\_derivatives\_prognostic(ground)

%put spatial derivative here

end

%> Descitption of the functions, input and output parameters

function timestep = get\_timestep(ground)

%put estimate for maximum timestep which allows stable forward Euler here

end

%> Descitption of the functions, input and output parameters

function ground = advance\_prognostic(ground, timestep)

%put advancing in time here

%real timestep derived as minimum of several classes in [sec] is used here!

end

%> Descitption of the functions, input and output parameters

function ground = compute\_diagnostic\_first\_cell(ground, forcing)

%put stuff that happens only if this cell is the first cell here

end

%> Descitption of the functions, input and output parameters

function ground = compute\_diagnostic(ground, forcing)

%put stuff that happens in every cell here

end

end

end

end

**Structure of the Parameter File**

In each class the variables are handled in different types: constants, parameter, state variables and temporal variables. The constants get defined in *CONSTANT\_excel.xlsx* globally for each run. Please use the naming convention detailed below. The parameter get defined in *yourRunningNumber.xlsx* seperate for each used class.

The parameters need to be filled in as follows:

CLASS index

GROUND\_yourName 1

value default unit

other\_parameter value default [unit] description

density 300 350 [kg/m3] snow density

albedo 0.8 0.8 [-] surface albedo

other\_parameter value default [unit] description

CLASS\_END

The file is scanned to the key words CLASS and than for pre-defined ("allowed") parameter names. Thus only the first three columns are read and the rest can be used for clarification and comments. Empty lines and lines with dashes can be used to devide the parameters. The index besides the class name allows for different modules based an the same class, e.g. two ground modules with the same functionalities but different albedo.

The same structure is used for parameter setting of the forcing, the grid and the stratigraphy.